506.1-506.3
Hardening Linux/Unix Systems Sections 1-3

SANS

THE MOST TRUSTED SOURCE FOR INFORMATION SECURITY TRAINING, CERTIFICATION, AND RESEARCH | sans.org

SEC506.1 Securing Linux/Unix

SANS

SEC506 – Securing Linux/Unix
Introductory Notes

Hal Pomeranz
hal@deer-run.com
http://www.deer-run.com/~hal/
@hal_pomeranz on Twitter

Who Is Hal Pomeranz?

Here at SANS:

- Track coordinator/primary "Unix dude"
- The oldest surviving SANS Faculty member

In my other life:

- Independent security consultant
- Wearer of many hats…

Welcome to the Unix Security Track! The instructor standing in front of you is probably Hal Pomeranz. Hal is the track coordinator course author for this Unix Security track, so he's the right person to contact with any feedback or questions on the material presented here (contact info on the previous slide).

Hal also has the (dubious) distinction of being the instructor who's "been around the longest," having taught his first tutorial for SANS in 1994 and actually presented a paper at the very first SANS conference (back before it was called SANS, in fact). He also wrote the original Unix material for SANS' introductory "Security Essentials" curriculum and was the maintainer of that material until 2007.

When not teaching for SANS, Hal works as an independent computer forensic investigator and security consultant, traveling all over the United States, and sometimes outside of the country. He is a co-author of the Command-Line Kung Fu blog (*http://blog.commandlinekungfu.com*) with SANS faculty members Ed Skoudis and Tim Medin. Hal also contributes to the SANS Computer Forensics blog (*http://blogs.sans.org/computer-forensics/*) and is one of the instructors for the SANS Computer Forensics curriculum.

What's in This Track?

Day 1-3 –Hardening Linux/Unix
- Day 1: Minimization, SSH, Firewalls
- Day 2: Rootkits and FIA, Access Ctrl, Kernel Tweaks
- Day 3: More Fun With SSH, Logging

Day 4-5 –Application Security
- Day 4: Tools and Methods + scponly Shell
- Day 5: BIND, Apache, ...

Day 6 –Unix Digital Forensics

If you're taking the entire track, it's helpful to understand the course progression logic…

Days 1-3 focus on hardening the basic Linux/Unix operating system. I also include some material on common exploits so that you understand what we're trying to protect against. While the first three days tend to flow together into a continuous curriculum, each explicitly covers several major topics. Day 1 is all about remote exploits and minimizing the vulnerability profile of your system by stripping down OS functionality, using encrypted access via SSH, and turning on host-based firewalling to limit network access. Day 2 starts by discussing the rootkits attackers will use once they gain access to your system and the file integrity assessment (FIA) tools you can use to detect this activity. Then Day 2 shifts into looking at access control issues and different "knobs" you can turn in the system to tighten things up. Day 3 focuses more deeply on monitoring your system and introduces a number of SSH "tips and tricks" that can be helpful when managing your systems.

But having a secure operating system platform isn't enough. You also have to pay attention to the applications you run on top of that platform. So, Days 4 and 5 focus on application security on Linux/Unix systems. Day 4 discusses the Unix `chroot()` mechanism (along with the useful scponly shell as an example of a `chroot()`ed application) and SELinux as tools for locking down any app. Day 5 covers specific security functionality in the most popular "internet facing" applications on Unix servers such as BIND and Apache.

Finally, on Day 6, we look at tools and techniques for investigating a compromised Unix system and figuring out "what happened" so that you can get your machine back online as quickly as possible.

We will only be covering a single section per day, so you don't need to lug all of the books around with you all the time. We will have "hands-on" lab exercises every day, so be sure to bring your laptops with you.

ASK QUESTIONS!!!

Feel free to interrupt with questions at any time. Trust me, if you're confused about something, it's likely that plenty of other people in the room have the same question. Instructors are also happy to answer questions during breaks.

SANS instructors love talking about the subject matter—we wouldn't be here otherwise. We want our students to get involved with the material and really dig into it. Asking questions is one way you can "customize" the course to fit your needs.

However, sometimes if the answer to the question isn't going to be relevant to many of the people in the class, the instructor may ask you to defer your question to one of the breaks so that the class won't get derailed on tangents. Break time is also a great time to dive down into minutiae.

Linux/Unix Hardening (Day 1)

Hal Pomeranz
hal@deer-run.com
http://www.deer-run.com/~hal/
@hal_pomeranz on Twitter

Much of the guidance in this course can also be found in the free OS Benchmarks available from the Center for Internet Security (*http://www.cisecurity.org*).

About This Course

- Focus on the OS-level security "knobs" in a typical Linux/Unix operating system
- Specific examples use RHEL Linux
- Use the CIS guides as a "Rosetta Stone" for other OSes

This curriculum takes a "bottom up" approach to Unix security, so we're going to start off by looking at the basic "knobs" that are available in most Unix-like operating systems for improving system security. Later, courses will also introduce additional security tools that may not be provided with the basic OS.

For purposes of the actual examples in the course, we went with a "market share" decision and will be using primarily RedHat Linux-based distros. But, wherever it's relevant, the course will also discuss how things are done on Debian/Ubuntu, BSD Unix, Solaris, HP/UX, AIX, etc. The important thing to remember is that the "knobs" we're looking at exist in various forms in nearly every Unix-like system available today. So, pay attention to the concepts more than the syntax—the "why" more than the "how."

As far as the "how" goes, there are a large number of Unix security "recipes" out there on the internet for many, many different operating systems. The free CIS hardening "benchmarks" at *CISecurity.org* are a good place to start if you want to "translate" the guidance in this course into actionable commands on different Unix and Linux variants.

This version of the course has been continuously updated and tested against from the early days of RedHat desktop operating systems all the way up through RHEL 7.x (CentOS 7.x). Much of the Linux guidance will actually apply to any Linux distro. I certainly use these techniques on my Linux Mint (Debian) laptop.

The Critical 80%

Minimize vulnerability exposure:

- Install minimal OS image
- Keep up-to-date on patches
- Disable/filter unused services/ports

Use encrypted access only

Intensive logging/monitoring

The question I get asked most often seems to be "*What are the most important things I can do to secure my Unix-like system*?" Assuming your biggest concern is external attacks, then the items on this slide are the critical tasks that will eliminate 80% or more of your external vulnerabilities (we'll actually be covering the first two major bullet items today, but an in-depth discussion of logging and monitoring will have to wait for Day 3).

The biggest component is to present as small a target as possible by reducing the number of potential vulnerabilities on your system. This process starts from the moment you install the system with the smallest, most "minimal" OS image that will allow you to successfully run your application. Patches need to be applied to the base OS image and then maintained for the lifetime of the machine. Perhaps the most important task is to disable services that are not being used so that you're not exposed to as yet unknown vulnerabilities in these services. Sometimes, you can't completely disable a service, but you can use a host-based firewall or some other IP-based access control to restrict which machines may have access to that service.

If you're going to have a secure system that you'll be accessing remotely, it's important that you use secure login and file transfer protocols. We'll be talking about SSH in some detail over the next few days.

Even with the best possible security, a break-in or at least an attempted break-in will occur at some point. So, you need to monitor your system closely, taking advantage of every available source of logging the system can provide. On Days 2 and 3, we'll also be looking at building a centralized logging infrastructure to more easily monitor all of the systems in your enterprise, and the use of file integrity assessment (FIA) tools to monitor changes to your system.

But What About Today?

- *Memory Attacks and Overflows*

Minimization and Patching

- *Session Hijacking*

Encrypted Access With SSH

Host-Based Firewalling

SANS SEC506 | Securing Linux/Unix 8

For today's class though, we're going to focus in on some of the basic, but also some of the most high-value techniques for protecting against external threats to your systems.

First, we'll begin with a look at remote exploits involving buffer overflows and other sorts of memory corruption attacks, since these are some of the most common remote exploits that you're likely to encounter in the wild. This will then lead into a discussion of minimizing the number of potential vulnerabilities in your systems by disabling unused services and patching.

We'll talk a little bit about session hijacking and do a little demo so you understand why using encrypted login channels like SSH is so important for maintaining the security of your systems. We'll look at some of the basics of SSH configuration today, and then on Day 3 we'll also look at some cool SSH-related "tips and tricks."

Finally, we'll finish up with a module on controlling access to your systems using host-based firewalling tools.

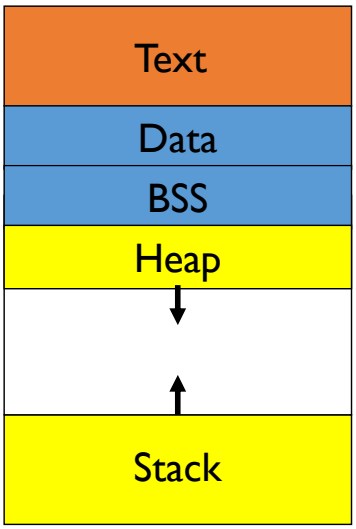
Memory Attacks and Overflows

Before we start tackling how to secure our systems, it's useful to talk about some of the ways in which those systems can be compromised. Memory attacks of various sorts—buffer overflows, heap overflows, return to libc attacks, format string attacks, and so on—have become an enormously popular mechanism for gaining remote access to systems.

The first widely known buffer overflow exploit was a fingerd exploit in VAX systems running 4.3 BSD. This overflow was one of the propagation mechanisms for the Morris Worm in 1988. The first paper on exploiting buffer overflows appears to have been published by Mudge circa 1995 and led to the popularization of the term "buffer overflow." One of the most readable, step-by-step buffer overflow guides appears in Phrack #49 (1996): "*Smashing the Stack for Fun and Profit*" by Aleph One. Back issues of Phrack are available from: *http://www.phrack.org/*.

This section attempts to describe how buffer overflows occur in a fashion that can be understood by non-programmers. Aleph1's paper contains actual exploit code and a more detailed analysis, and we'll be including links to more modern exploits as we go along (interesting reading if you have the programming background).

Process Memory



- ← Program instructions (read-only)
- ← "Constants" – literal strings/numbers
- ← Global and persistent (static) variables
- ← Dynamically allocated data
- ← Free/unallocated memory
- ← Local subroutine data

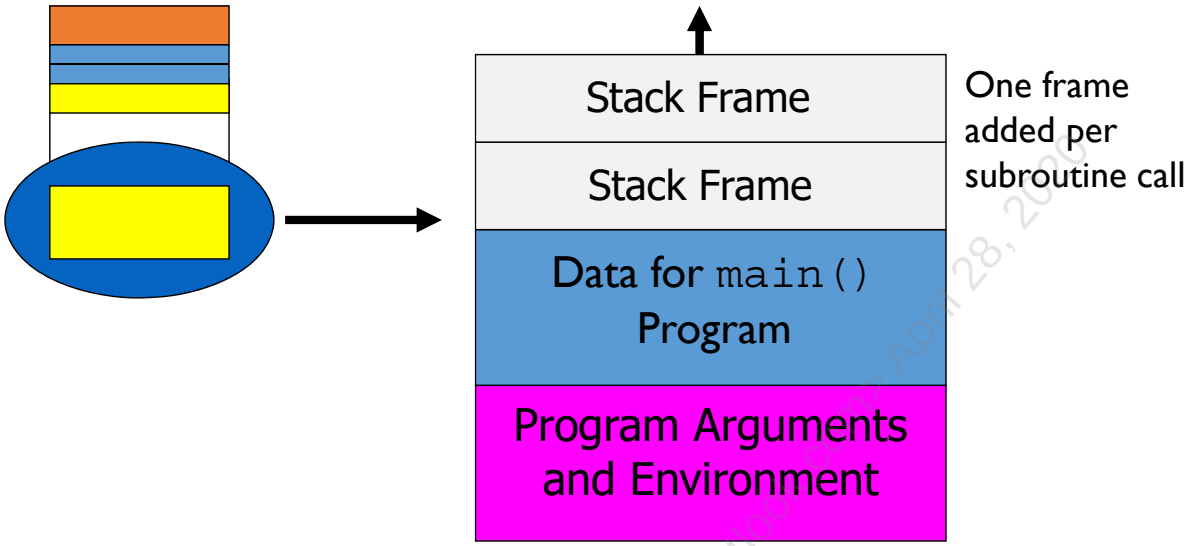
When a process is started on a Unix system, the Unix kernel gives it a "virtual" address space of its own. This memory space is divided into regions called *segments*. At the "bottom" of this memory space (closer to the lower memory addresses) is the actual code for the application—instructions, static data compiled into the program, etc. This memory area, referred to as the *text segment*, is of fixed size and read-only.

"Above" the text segment in memory (moving toward higher memory addresses) are the *data* and *BSS* (for "Block Storage Segment" and sometimes "Block Started by Symbol") *segments*. The BSS segment stores "global" variables (variables that are declared at the uppermost level of the program and are visible in any part of the code) and special variables whose values are persistent across subroutine calls (if you're a C programmer, these are the variables you declare as "`static`"). The data segment is where constant values used in the program are stored. To illustrate the difference between data and BSS, suppose your program had a global string variable called `mycommand`, and you assigned it the literal value of "`/bin/sh`". The data space for the `mycommand` variable would live in the BSS segment, but the actual string "`/bin/sh`" would be allocated in the data segment. The data and BSS segments are fixed size and do not grow/shrink during the lifetime of the program.

The next item in memory is a dynamic memory area referred to as the *heap*. The heap is where the program allocates space for dynamic variables that are allocated during program execution (for those of you who are C programmers, the heap is where the `malloc()` system call allocates data). The heap grows and shrinks dynamically as variables are allocated and discarded.

Starting at the "top" of the memory space is another dynamic data area referred to as the *stack*. The stack is the data area allocated to variables that are local to the various subroutines in the program. The stack also contains the command line arguments the program was invoked with as well as a complete copy of the environment the program was executed in. The stack's memory space grows and shrinks dynamically, but generally grows "downward" from the top of memory toward the heap area that is growing "up" from the bottom.

The Stack

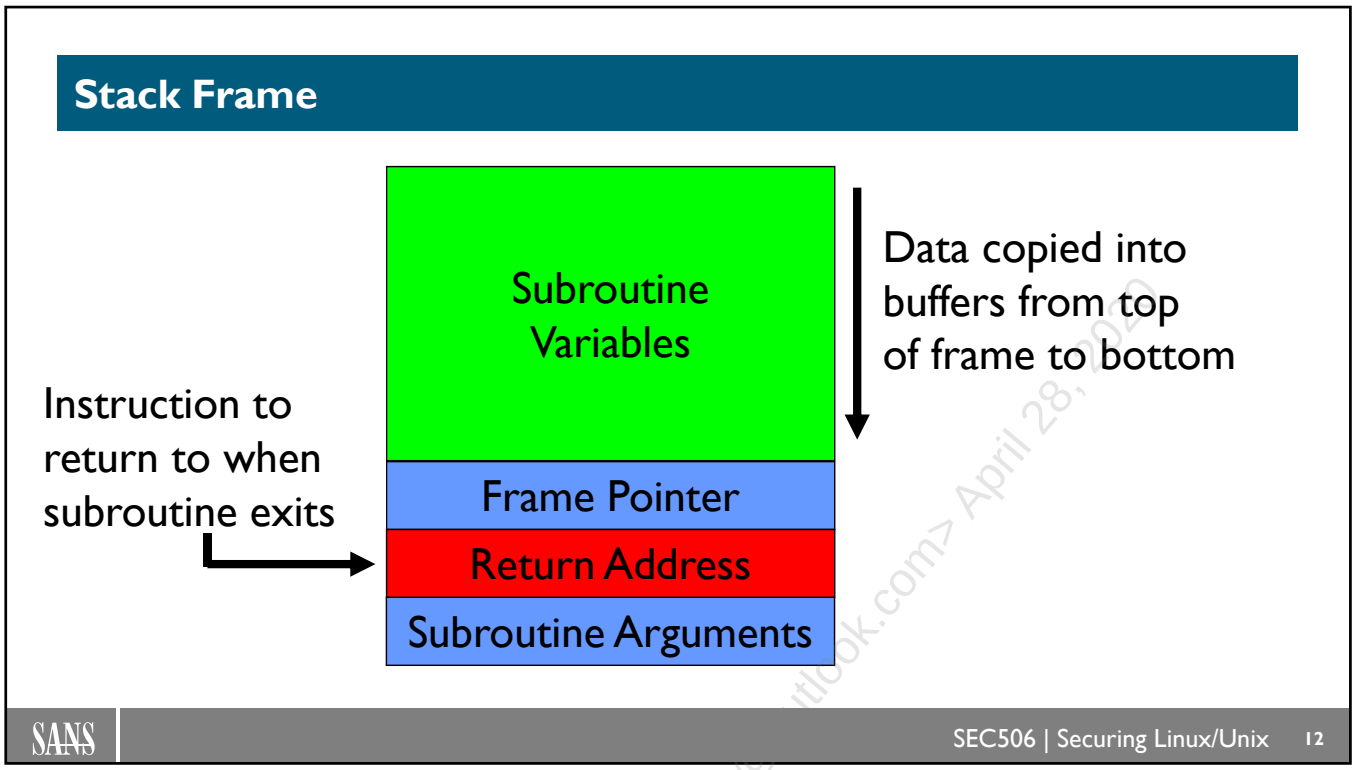


SANS | SEC506 | Securing Linux/Unix 11

We'll return to the rest of process memory later in this section, but for now let's focus in on the stack area. The stack takes its name from a particular type of abstract data structure. Essentially, the stack can be thought of like one of those spring-loaded stacks of plates you see in a cafeteria; you can pick the top plate off the stack or load more plates onto the top of the stack, but you can't get at the plates underneath ("last in, first out" or LIFO is another way to describe this behavior).

At the bottom of the stack are the environment variables at the time of the program's execution, plus any command line arguments specified by the user when the program was started. Above that is the data area allocated for the `main()` subroutine. In a C program, the `main()` subroutine is where program execution begins.

Every time the `main()` routine makes a subroutine call, a stack frame is pushed onto the top of the stack (more about the contents of a stack frame on the next slide). If that subroutine calls another subroutine, then another stack frame is pushed onto the top of the stack, etc. Eventually, each stack frame is "popped" back off the stack as each subroutine exits.



Looking at an individual stack frame, there are four components (bottom to top in this picture):

1. The arguments passed into the subroutine
2. An instruction address at which the program should resume execution when the subroutine exits, usually referred to as the *return address pointer*
3. A *frame pointer* used to index data in the stack frame (don't worry about this)
4. Enough space to hold all of the local variables used by the subroutine

Let's now suppose that one of the local variables in the upper portion of the stack is a string. It turns out that the first character in the string is closest to the top of the stack frame, and that other characters are written "downwards" toward the bottom of the stack frame. A buffer overflow exploit attempts to write past the end of the allocated string buffer and clobber the return execution address. If successful, the program will not return to the original calling function but will instead jump to some other malicious instruction created by the attacker.

Attackers overflow buffers by sending the program long input strings (via input prompts in the program, environment variables, Remote Procedure Calls, etc.). These attacks succeed because many programmers don't always check the length of the input they receive and happily accept long inputs, which can be used to overwrite memory in this fashion.

Classic Buffer Overflow (1)

Attacker constructs an input containing:

- No-ops for padding
- Machine code to `exec("/bin/sh")`
- Bogus instruction addr pointing into subroutine data area

Subroutine is coerced into copying this input into its data area, overwriting end of buffer

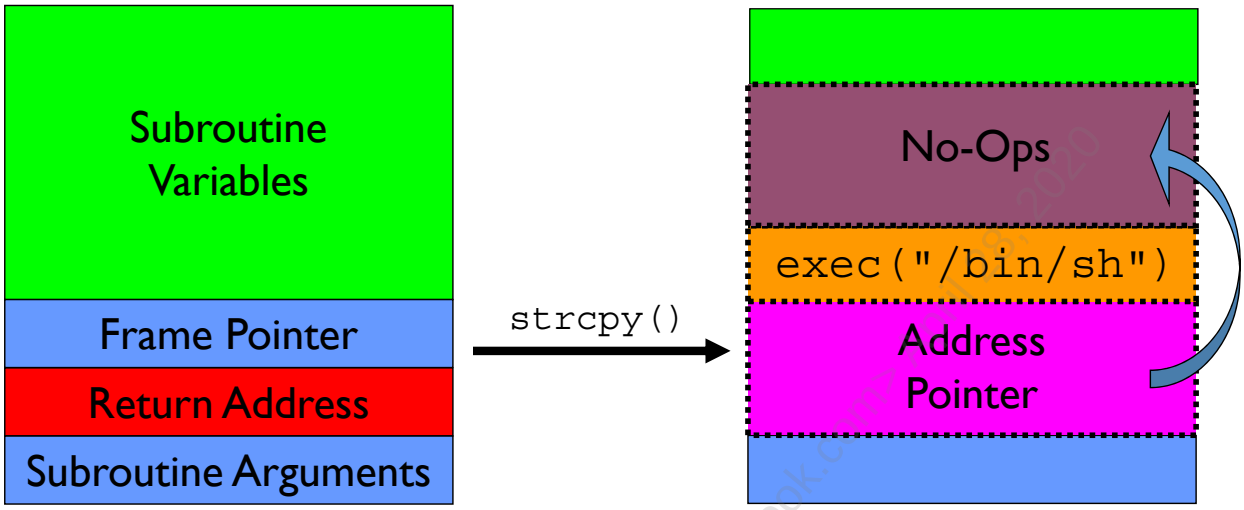
Subroutine exits and the program follows bogus address to execute shell

If program runs as root, attacker gets root shell

So, the question becomes, where does the attacker put their malicious code? Why, in the long string they use to clobber the stack frame, of course! The classic buffer overflow attacker constructs a string that contains a bunch of "no-op" instructions for padding, then the nasty code, then the evil instruction address repeated many, many times. The instruction address is repeated because the attacker can never really be exactly certain where the return pointer is relative to the buffer they're overflowing. The "no-ops" are added at the beginning of the string because the attacker can't be certain exactly where their evil instructions are going to be placed in memory. They hope the instruction address they concoct will jump into the middle of the "no-ops" and the program execution will eventually reach the evil instructions (it turns out the attacker can predict the memory address of the evil instruction set fairly closely, so this usually works).

The most common evil instruction set is a variant of the `exec()` system call, which replaces the current process with a new process. The "classic" buffer overflow described in Aleph1's paper uses `exec(/bin/sh)`, which gives the attacker an interactive shell, but newer attacks will typically run a more complicated script that downloads rootkits and possibly additional exploit code to compromise the system and make it into part of a botnet for DDoS attacks or spamming. Note that whatever new process the attacker runs has the privileges of the original process that was compromised by the attacker. Ideally, the attacker hopes this is root privileges, but they may be able to use a secondary privilege escalation attack to gain root privileges after compromising the original process if that process was running as an unprivileged user.

Classic Buffer Overflow (2)



Here's the picture: On the left, we have our normal stack frame right before the attacker sends their malicious input. On the right, we see the stack frame after the attacker has managed to overflow a buffer (`strcpy()` is a library routine that is famous for blindly copying string data without checking that the target buffer is big enough to accept the input).

Starting at the top of our picture of the stack frame, we see the area of "no-ops", then another chunk containing the evil code, and finally an area where the attacker just repeats the same 4-byte return address over and over again. Note that the `exec("/bin/sh")` code fits in 64 bytes of data or less (obviously more complicated exploit scripts will consume more space), but no-ops can be added to make the buffer as long as necessary to clobber the return address pointer.

If the attacker successfully clobbers the return address, then when the subroutine exits, the program execution jumps someplace into the no-op area. No-ops are executed, and then the evil instructions are reached and the root shell is produced.

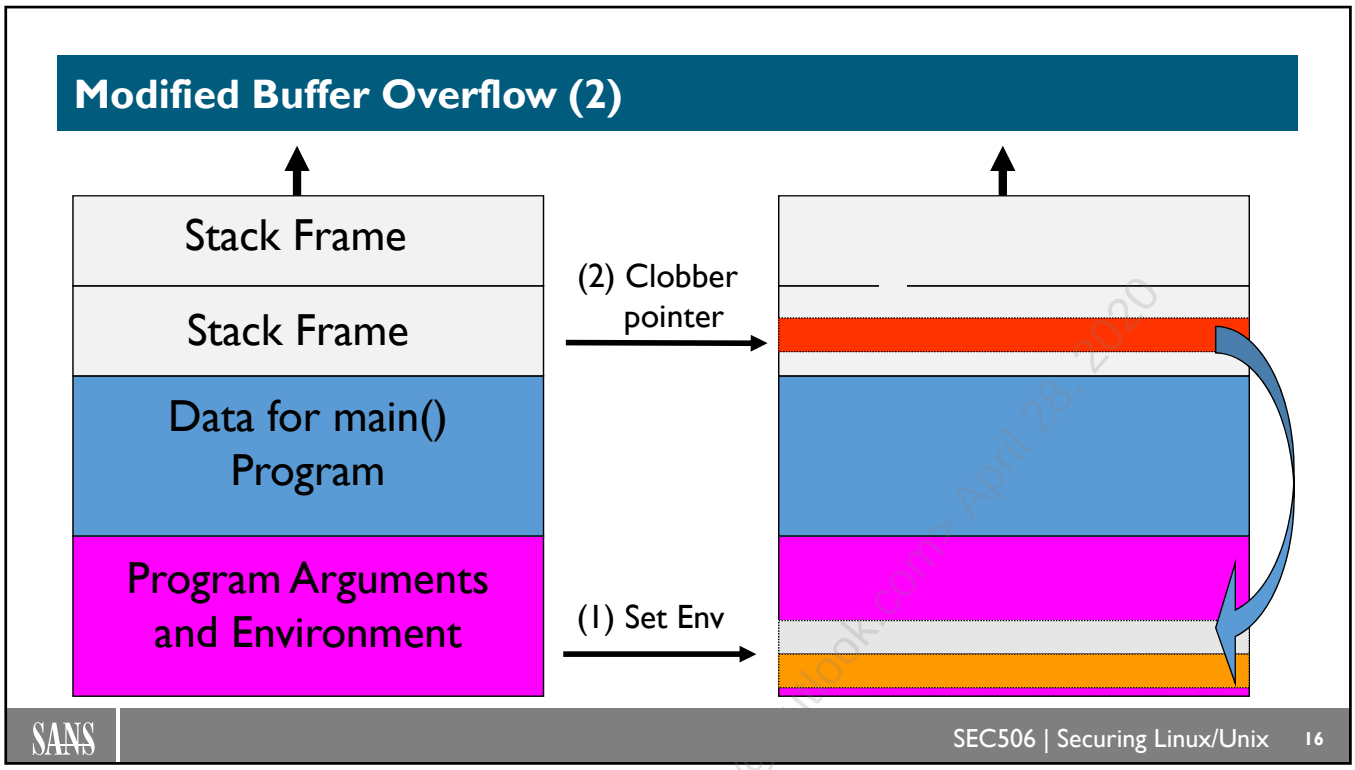
Modified Buffer Overflow (1)

- Sometimes, subroutine buffer is too small to hold exploit
- Exploit can be put into an environment variable
- Subroutine return address set to environment area at bottom of stack
- Requires attacker have local machine access ...

Sometimes, a program contains a buffer that can be overflowed but which isn't big enough to contain the malicious code. In this case, the easiest thing for the attacker to do is to put the malicious code into the program's environment before the program is executed. The attacker then resets the subroutine return address to jump into the bottom of the stack where the program's environment variables are kept. If the program isn't careful about checking its command-line arguments, then the malicious code could even be fed in as part of the command line used to invoke the vulnerable program.

Note that messing with the program's environment or command-line options usually means that the attacker has to be on the local system and sometimes execute the program themselves in the hostile environment (although there are some programs like good old `telnetd` that will accept environment variables from a remote client opening themselves up to remote buffer overflow exploits). On the other hand, if this is a set-UID root program, the attacker can use this sort of attack to get root privileges.

Later, we'll also discuss attacks where a remote attacker might put the malicious code into the heap area or some other part of program memory.



Here's a picture of this modified buffer overflow attack. The no-ops and malicious code are down at the bottom of the stack with the other environment variables, but the attacker is clobbering the return address pointer in one of the stack frames higher up in the stack once the program has begun executing.

In General …

Attacker places exploit in memory, changes address pointer to jump to exploit

Places to put exploit:

- Stack (per previous examples)
- Heap
- Data/BSS (if writable)

Ways to modify address pointer:

- Stack overflow (per previous examples)
- Format string attacks (see next slide)

So, in order to succeed, the attacker needs to be able to do two things:

1. Get malicious code into the memory space of the application.
2. Modify a return address pointer to jump to the malicious code.

As the last attack clearly shows, the malicious code doesn't have to live in the stack frame. You can put it anyplace in memory and adjust the return pointer in the stack frame accordingly. In fact, the malicious code doesn't even have to go into the stack area at all. If the attacker can overwrite a dynamic data structure that lives on the heap, that will work fine. Variables in the BSS could also be used to hold the malicious code.

Aside from finding new places to put their malicious code, attackers are also finding different ways to clobber the return address pointer in the stack frame…

Format String Attacks

- Programmers are sometimes lazy:

```
printf("%s", str);        # correct
printf(str);              # lazy
```

- If attacker can set `str`, attacker can set output specifiers
- `%n` writes a numeric value to a specified memory address
- Attacker can overwrite return address pointer

One method for overwriting the return address pointer is a format string attack. Unfortunately, it requires a reasonable amount of programming background to understand what's happening, so we won't discuss format string attacks in detail here. Interested folks are referred to Tim Newsham's white paper on the subject at: *http://seclists.org/bugtraq/2000/Sep/214*

When a C program wants to print a string, the correct practice is to use the first `printf()` call shown above. The first argument to the `printf()` routine (`"%s"`) is the format string that specifies what should be printed. This format string can contain literal text or special markers like `%s` that are used for variable substitution. For example, `"%s"` says to print the string variable that is the next argument to the `printf()` call.

However, the second call also works because the `printf()` routine just interprets the string variable as the format string and prints out the contents of the string. But what if the variable `str` contains things like `%s`, `%d`, etc.? The `printf()` statement would just start randomly printing values off the stack as "arguments" that needed to be printed. The problem is that `printf()` doesn't just print things out: `%n` inside a `printf()` format string causes a numeric value to be written to a memory location given to the `printf()` routine as an argument.

The upshot is that if the variable str is taken from user input, the attacker can construct str in such a way that an arbitrary numeric value gets written to an arbitrary memory address. This means that the attacker could overwrite the stack frame return instruction pointer so that it jumps to some other memory area containing the malicious code.

Fixing Buffer Overflows

Fix the programs
- Too many
- New bugs being written daily

Fix the programmers
- Too many
- New programmers being made daily

Fix the stack

So, what do you do? You could hire an army of programmers to fix all of your problems. This is great if you have the resources and can find enough programmers. However, even the best programmers make mistakes, which is why we're having these problems in the first place.

It turns out that thinking about this problem a little yields a better alternative.

Fixing the Stack

- Program code normally resides in read-only text segment
- Programs should never execute instructions from stack (or any other writable data segment)
- Modify kernel—attempts to execute from stack cause program to abort
- Requires cooperation of CPU hardware

For any normal program (one that doesn't use self-modifying code) program instructions always reside in the read-only "program text" area at the beginning of memory. Well-behaved programs should *never* execute code in any other portion of memory, e.g., from the stack or heap areas. The "fix" for most buffer overflow attacks is to simply modify the kernel so that attempting to execute an instruction in the stack area causes the program to abort. This behavior is generally referred to as *stack protection*.

Note, however, that stack protection generally requires the cooperation of the memory management unit on the system CPU. Historically, proprietary CPUs (Sparc, Alpha, HP RISC, etc.) provided the right hardware hooks for the OS kernel. The "commodity CPUs" (older 32-bit Intel processors and PowerPC) generally did not. This meant that stack protection first appeared in the proprietary Unix flavors like Solaris while Linux/BSD lagged behind. However, OpenBSD implemented a kernel-based stack protection solution called W^X in OpenBSD 3.4, and the grsecurity patches to the Linux kernel (see *grsecurity.org*) included the PaX stack-protection implementation from the Admantix Linux project. Red Hat has included their own "Exec Shield" implementation in the kernel since Red Hat Enterprise 3. However, these were all software-only implementations and therefore less efficient.

64-bit Intel and AMD chips have the appropriate hardware support on board. This is generically referred to as "NX bit" support after the AMD64 implementation. Pentium-4 (and newer) 32-bit CPUs also include NX bit functionality. The Linux kernel has included support for NX bit functionality since version 2.6.8, and it's on by default (boot the kernel with the `noexec=off` argument to disable the feature).

If you're interested in further technical details, a good place to start reading is:
http://en.wikipedia.org/wiki/NX_bit

Another good resource is this older presentation discussing security changes in OpenBSD 3.x, particularly the notes on combatting memory attacks and overflows: *http://www.openbsd.org/papers/csw03.mgp*

Different Implementations

- For Solaris, HP-UX, et al:
  - "Stack protection" only prevents executing code off of stack pages
  - Can be thwarted by putting malicious code into the heap area
- Linux (NX and PaX), OpenBSD (W^X):
  - All writable pages marked as non-executable (including heap area)

It turns out that different vendors provide different coverage as their stack protection solutions go.

Solaris and HP-UX take a conservative approach—these stack protection implementations will only abort programs that attempt to execute code in the stack areas of the program (Sun chose this implementation because the early Java runtimes would execute code off the heap). This means that attackers can thwart these measures simply by getting their malicious code into the heap.

On the other hand, OpenBSD's W^X implementation and the standard Linux implementations (both the NX bit support in 2.6.8 and later kernels and the PaX implementation used in the Linux grsecurity patches) refuse to execute instructions out of any writable memory page, which includes the stack, heap, and data areas on some architectures.

It's always good to test your application with the stack protection solution for your operating system platform. Anecdotally, across all of my customer's machines over the decades when I've been working with Unix and kernel-based stack protection, I've only ever found one application that fails to operate when stack protection is enabled. In that one case, the vendor acknowledged that it was a bug in their application and produced a fix in the next release.

Enabling Stack Protection

On by default in Linux (since v2.6.8)

Want some validation?

```
# dmesg | grep NX
NX (Execute Disable) protection: active
```


Stack protection is enabled by default in Linux since kernel version 2.6.8—at least on 64-bit CPUs. There was a time in the distant past when Linux vendors would disable stack protection in their 32-bit kernels for backwards compatibility reasons, but these days every kernel should have stack protection enabled by default.

You can always check to make sure that stack protection is enabled using the command on the slide. If you `grep` through the boot messages from your system for the keyword "NX", you should see an indication that stack protection is active. If you must disable it for some reason, you can boot your kernel with the `noexec=off` option.

For Solaris machines, you can enable stack protection with the `/etc/system` settings:

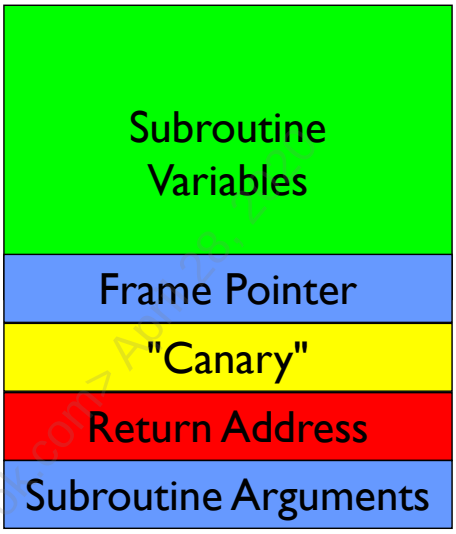
```
noexec_user_stack = 1
noexec_user_stack_log = 1
```

`noexec_user_stack` enables stack protection. The `noexec_user_stack_log` setting causes the kernel to log to syslog when it shuts down an application that is violating the stack protection rules. Note that these settings are actually the default for modern Solaris operating systems, although it never hurts to explicitly set them in the `/etc/system` file.

As far as other operating systems go, OpenBSD enables stack protection by default. On HP-UX systems, you have to enable stack protection yourself with the "`/usr/sbin/kmtune -s executable_stack=0`" command.

Compiler-Based Solutions

- Compiler inserts random "canary" value before address pointer
- Canary value checked when subroutine exits—abort if changed
- Attacks that clobber return address pointer also clobber canary
- Performance hit due to extra code to insert/check canary values



While kernel-based stack protection can stop buffer overflows in any application running on the machine, work is also being done to develop compilers that produce executable code that is resistant to buffer overflows. This is of particular interest on platforms where kernel-based stack protection is not available but is also used in conjunction with normal kernel-based stack protection.

The idea is that an additional data value called a "canary" (as an analogy to the canaries coal miners use to warn them of toxic gasses) is inserted into the normal stack frame between the return address pointer and the subroutine variables area. Any classic buffer overflow exploit that overflows the data area and writes downward to the return address pointer is also going to overwrite the canary value (although this is not true in a format string attack). As part of the normal subroutine exit sequence, the canary value is checked. If the value changes, then the program simply aborts itself rather than returning to the memory address in the (probably corrupted) return address pointer.

Obviously, there is some performance overhead in inserting the canary value into each subroutine and then checking the value again when the subroutine exits. Applications with this canary-based stack protection feature seem to execute about 10% slower than "normal" uninstrumented applications.

This kind of stack protection is available using the `-fstack-protector` and `-fstack-protector-all` options in GCC (the GCC stack protection support is based on the IBM's "ProPolice" implementation). OpenBSD has compiled all of its packages with stack protection since as far back as OpenBSD 3.3, and SUSE has compiled all packages with stack protection since OpenSUSE 10.3.

How to Defeat Stack Protection

Heap exploits

Force subroutine return to legitimate call, like `system()`

Overwrite expected subroutine args:

- `system()/exec()` call wrong program
- `open()` opens unexpected file

It is important to know that the stack protection solutions we have covered so far are not perfect. People have been figuring out how to get around these sorts of protections in different ways.

We already mentioned how the Solaris and HP-UX stack protection implementations can be thwarted by putting the malicious code into the heap area. However, there are deeper kinds of heap exploits that are beyond the scope of this course. An interesting tutorial can be found at:

http://www.mathyvanhoef.com/2013/02/understanding-heap-exploiting-heap.html

In addition, an older white paper on the subject is available at *http://www.cgsecurity.org/exploit/heaptut.txt*

Another approach would be to overwrite the return address pointer at the bottom of the stack frame, but instead of using a memory address in the stack or heap area, the attacker causes the program to jump to the memory address of some other subroutine in the text segment, a so-called *"return to libc attack"* (sometimes called a *"libc redirection attack"*). The program is now returning to a "valid" memory address in the "safe" part of program memory, so kernel-based stack protection won't help here. But, suppose the attacker causes the program to return to a library call like `system()` that executes some other arbitrary program with the privileges of the original process. This could allow the attacker to open a back-door onto the remote system. For more information, see *https://css.csail.mit.edu/6.858/2014/readings/return-to-libc.pdf*. Compiler-based stack protection might help here if the attacker clobbers the return address pointer with a standard overflow type attack in the stack frame. However, an attacker might be able to "miss" the canary value by using a format string attack instead.

Maybe the attacker doesn't need to modify any return address pointers at all. Suppose the attacker discovers a buffer overrun in a subroutine that calls `system()`, or even just `open()` to read/write a file. If the attacker can overwrite the arguments to one of these calls, they can cause the program to do unexpected things. For example, `system()` might be coerced into calling a dangerous program, and `open()` could be forced to open a file like `/etc/shadow` and make changes.

Lab Exercise

- Launch an exploit with Metasploit
- Understand how to use Metasploit for testing purposes

Exercises are in HTML files in two places: On the course USB media and in the home directory of the SANS user in the lab virtual machine. These two locations contain identical copies of the exercise files; it just depends on whether you prefer to look at them from a web browser inside the lab VM (because you're working in full-screen mode) or from your host operating system.

1. Open your web browser
2. Select "File… Open…" from the browser menu (Ctrl-O is the usual keyboard shortcut)
3. This is Day 1, Exercise 1, so navigate to *.../Exercises/Day_1/index.html* and open this HTML file
4. Click on the hyperlink, which is the title of Exercise 1
5. Follow the instructions in the exercise

Minimization and Patching

Now we're going to turn our attention to ways to help protect against remote exploits. First up is minimizing and patching your OS and the services that run on it in order to reduce your overall system "vulnerability profile."

What Do You Do?

Hierarchy of mitigation:

- Disable service
- Patch known vulnerabilities
- Use strong encryption
- Filter network access
- Isolate application
- Run unprivileged

Hiding app versions can help somewhat

There's a lot of advice out there on how to protect yourself from vulnerabilities like we've seen so far and all of the new vulnerabilities that are being reported every day. When you distill all this advice down, however, it all boils down to the following simple rules:

- Disable all services that are not absolutely necessary for accomplishing your mission. The best way to prevent an attacker from exploiting a vulnerability on your system is to not be running the vulnerable software in the first place.
- Keep up-to-date on security announcements and patches from your vendor. Don't get victimized by vulnerabilities that have been known for months. Most vendors have security alert mailing lists you can subscribe to for timely updates.
- Use strong encryption to protect the data flowing between client and server. Strong encryption is also the only real protection you have from session hijacking.
- Only grant access to systems that have a legitimate need to access a given service. You wouldn't give copies of your car key to anybody who wants to take your car for a little spin, so why do you let them test drive your web server at will?
- As much as possible, try to run "one app per server" so that a vulnerability in one service doesn't end up compromising another. Some applications also allow administrators to run them in a restricted environment via the `chroot()` system call (more on this later in the curriculum).
- Where possible, avoid running applications with root privilege, so that if an attacker does get access to the system, there are some limits on what they can do. Kernel-based privilege restriction solutions like SELinux are also helpful here.

Removing OS version information from login banners can help a little as far as avoiding well-known vulnerabilities (security through obscurity doesn't work as a general security posture, but obscurity is a layer that can be used to help protect systems). Be aware, however, that scanning tools (e.g., Nmap) and automated worms are becoming more sophisticated every day.

Disable vs. Patch

- Patches protect you from problems we already know about
- Disabling services protects you from undiscovered exploits

If you don't need it, turn it off!

If you're not sure, turn it off and see what breaks!

We install patches from our vendors to fix problems we already know about. We disable services to protect us from problems we haven't discovered yet. Nothing feels better than reading about the latest Apache, SNMP, FTP, or whatever vulnerability on BUGTRAQ and knowing that you're not vulnerable because you don't run Apache, SNMP, FTP, etc. Which is not to say that you shouldn't go get the appropriate patches from your vendor—after all, you might end up running one of these services in the future, but you don't have to panic right away or wonder if somebody used the exploit to compromise your systems before it became public knowledge.

When disabling services, my best advice is, be paranoid. Unless you absolutely have to have a given service enabled, turn it off, even if you can't plausibly construct a way it could be used to exploit your system. If you're not sure what a given service does (and you can't figure it out from the manual pages) turn it off and see if anything breaks. You can always turn it back on later if there's a problem. However, you probably don't want to be "experimenting" like this on your production database machine. So, try these kinds of things out on a test machine or your desktop first.

OS Minimization and Patching

It's time to get the hardware together, configure the system, partition the drives, and install the smallest possible OS image. Reduced size implies reduced complexity and greater security, but also a loss of convenience. Choose security over convenience. If in doubt about the necessity of some service, turn it off and see what breaks.

Once you've got your OS installed, you need to apply your first set of patches. But patches need to be continuously maintained over the lifetime of the system.

OS Installation Choices

Pick the smallest possible OS image:

- Do you really need X Windows?
- Do you need LDAP, NFS/Samba?
- Do you need compilers/development tools?

Smaller doesn't just mean more secure:

- System is more stable (less to go wrong)
- System boots faster (less stuff to start up)

From a security perspective, you want to choose the smallest set of OS programs possible for your application because each piece of software you add might have a security vulnerability that could be used to exploit your system. It also turns out that smaller OS images are also a win from a system stability perspective (less that could go wrong) and from a performance perspective (less stuff is running and consuming system resources, and the system boots much quicker if it doesn't have to start dozens of different daemons at boot time).

Picking the smallest possible set of OS components is inherently site and application specific. All Unix systems will need a base operating system image: The standard Unix kernel and hardware drivers for disk drives, network interfaces, etc., plus the standard Unix command shells (`sh`, `bash`, `ksh`, `csh`, etc.) and the normal Unix command-line-oriented commands (`/usr/bin/*`, `/usr/sbin/*`, etc.). Beyond that, each site needs to decide:

- Does the machine need to run X Windows? If the machine is a server that's only ever going to be managed via the network, then you probably don't need to install all those GUI applications.
- Does the machine need all of those "workgroup" type sharing tools; LDAP (or NIS/NIS+ if you're "old school") and remote file services like NFS or Samba? For a desktop machine on a protected network, probably you want these tools, but not on a web server directly connected to the internet.
- How about compilers and other development tools (`make`, `ld`, `ar`, etc.)? On a developer's desktop system, sure. On other machines, though, they just take up space and encourage people to "do the wrong thing" and develop code on non-development systems. There's an argument that says these tools also make it easier for attackers to bring exploits onto your system, but since most exploits these days are precompiled binary packages, I don't agree with this argument.

The Problem with "Minimal"

Software vendors generally test against the "full" OS install:

- App doesn't run under "minimal" install because of missing packages
- Vendor has no idea what the "minimum" set of required packages is

Takes some work to figure this out for yourself

Have a system with a full install handy...

The problem you run into when you use a "minimal" OS image is that the applications you want to run on top of that OS image frequently won't work. It seems to be common practice among software vendors (and even the OS vendors) to only test their software on the "full" install of a given OS. When you try and make the system more secure by eliminating "unnecessary" components, you discover that there's some weird hidden dependency with the application that you're trying to install. The software vendor is no help. They are usually not prepared to tell you the "minimum" collection of OS packages necessary to run their software.

So, you get to figure out the packages you need through a process of iterative failure, which is described on the next slide. Note that when going through this process, it helps to have a second machine with a full OS install handy, or at least the package database from a machine with a full install.

What Packages Do I Need?

1. Attempt to install and run app
2. Determine missing file(s) via error messages/`strace`
3. Get package name from package database
4. Add appropriate package
5. Add appropriate dependencies
6. Repeat Steps 1-5 until done

The first step of figuring out what packages you need is to attempt to install and run the application. In some cases, even the install program will fail due to missing packages. Other times, the application installs fine but fails to run or quits after a short period. If you're lucky, there will be an error message (either in the logs or on the standard output) that will give you a clue about the missing file. If not, then you'll have to run the application under `strace` to see what file it was trying to open when it failed. We'll see an example of `strace` later in the course when we talk about application security and `chroot()`.

Once you know the file that the app is looking for, go find what package the file belongs to by looking in the system package database on your machine that has the full OS install. On Red Hat systems, use "`rpm -qf <filename>`" to find out what package a given file belongs to. Note that you must specify the full pathname of the file to "`rpm -qf`" ("`rpm -qla | grep <filename>`" can help you discover the full pathname if you're unsure). "`dpkg --search <keyword>`" is the equivalent for Debian Linux (including Ubuntu, Linux Mint, etc.). You don't have to use full pathnames here.

Once you know the appropriate package(s), you can manually install these packages off the OS CD. It's "`yum install`" to install packages on Red Hat systems, or "`apt-get install`" on Debian. A given package may also have dependencies, which means other packages must be installed before you install the package you want. The installer should automatically install these dependencies.

Note that you may have to repeat this process several times until the application is fully working. The whole process can be time-consuming and frustrating. The good news is that you generally only have to do it once per application. After the initial setup, you can just replicate your configuration to other systems needing to run that app. Also, after going through this process, you may want to reinstall the system from scratch with the packages you need, just so you have a clean "baseline" image to go forward from.

The Tao of Patches

- Only install patches *after* installing required OS packages
- Patches must be continuously updated
- Test patches on non-production systems before release
- Look into automation strategies

It's important to install patches only after you've completely determined which OS packages you need and have installed all of those packages. If you were to install part of the OS, apply patches, and then go back and add more OS packages, you'd end up in a situation where you had unpatched packages installed on the machine.

Of course, just installing patches once when you install the system for the first time isn't enough. You have to regularly download and install the latest patches from your vendor; particularly security patches. Having some sort of automated mechanism for doing this is often the difference between keeping patches up-to-date and never patching systems at all. If you rely on administrators to do this manually, they won't do it.

Of course, each patch you add has the risk of breaking your system in some way. For this reason, it's a good idea to first test patches on some non-production systems before rolling a patch out across all of the machines in your environment.

Red Hat Patching

YUM is the standard updating system:

- Open Source utility
- Easy to set up local repositories
- Built on top of RPM

"Rollback" feature removed as of RPM v4.6.0 (RHEL 6.x)

YUM is (in most respects) a perfectly adequate package update system and, in fact, is better than most. One of the problems with many updating systems is that you have to pay your vendor a lot of money for their special software if you want to set up your own local patch repositories. YUM makes setting up multiple repository servers very simple, which is particularly important if you want to set up an update server for machines that are not internet-connected.

Once you start making heavy use of YUM, you'll probably want to check out Extra Packages for Enterprise Linux (EPEL). See *http://fedoraproject.org/wiki/EPEL*. These are additional Open Source packages from the Fedora project that you can import into your OS. On CentOS, you can "`yum install epel-release`" to make this repository available in your OS.

Note that the RPM updates for Red Hat Enterprise releases are not freely available; you have to pay a subscription fee to the Red Hat Network to get them (your first year's subscription is included in the cost of the OS). On the other hand, once you're subscribed to the Red Hat network, you do get a fairly nice web-based management console for configuring, initiating, and tracking patch updates on all of your machines, so it's not like there's no extra value there.

The Ubuntu project has put together a nice document showing the mapping between RedHat Linux interfaces and Ubuntu (Debian) Linux interfaces. In particular, there's a nice chart mapping `yum`/`rpm` commands in RedHat to the equivalent `apt`/`dpkg` commands in Ubuntu:

https://help.ubuntu.com/community/SwitchingToUbuntu/FromLinux/RedHatEnterpriseLinuxAndFedora#Package_Management

One patching feature that's common to most of the proprietary Unix OSes (e.g. Solaris), however, is that you can always "roll back" patches that break functionality and get back to the pre-patched state of your system.

Unfortunately, the model for typical Linux distros is "always forward, never back" when it comes to software updates. This is one of those issues that Linux detractors point to when they say Linux is not "enterprise ready."

The RPM package management system that YUM is built on top of did at one time have the ability to save "rollback RPMs" before doing an update. Unfortunately, this feature was never implemented very well and was typically unreliable. Rather than fixing the problems, the maintainers of the RPM software decided to simply drop this functionality as of v4.6.0—which is the version of RPM that went into RHEL 6. See *http://www.rpm.org/wiki/Releases/4.6.0#Removedfeatures*

If the ability to roll back patches is important to you, you can emulate this functionality in a couple of different ways:

- In virtual machine environments, just take a snapshot immediately before applying updates. If things don't work out, revert to the snapshot.
- Mirror your root drive, but sync the filesystem and break the mirror before applying updates. If the update doesn't work, boot off the original version of the root drive in the off-line mirror.

However, having a good testing environment so that you can certify patches before rolling them out to your production systems is obviously very important.

Stopping Boot Services

Even with a minimal OS install, most Unix systems ship with some default services that can and should be disabled for security.

Consider Disabling…

- NFS, Samba, automounters, etc.
- RPC services, portmapper
- Servers: Apache, **`lpd`**, SNMP, etc.
- "Conveniences": Power/Volume managers
- GUI logins and other unused login channels

There are lots of different classes of potentially dangerous services out there:

- File sharing via NFS and Samba is certainly dangerous for internet servers and should be disabled.
- RPC-based services are also incompatible with security in hostile, unprotected network environments. Many worms and exploits seem to have targeted vulnerable RPC services.
- Apache is fine if you're a web server, but if you're not then why run `httpd`? Disable printing (CUPS or `lpd`) if you don't print, SNMP if you're not monitoring your machines this way, etc.
- I find many of the "convenience" features like power management and volume managers to be more annoying than helpful. Volume managers also make it easy for users and attackers with physical access to compromise your system with tools and set-UID programs brought in on USB and DVD.
- If you only manage the system over the network, do you need that GUI login running on your machine's console (and the X server that goes with it)?

For specific recommendations regarding which services to disable, I suggest downloading the appropriate CIS Benchmark document for your OS from *http://www.cisecurity.org/*. While we're going to talk about managing a number of different services in some detail, each OS has its own particular set of issues that need to be addressed. Unfortunately, every OS update seems to change at least a few services—adding some and deleting others.

Traditional Boot Service Control

Service start/stop scripts in **`/etc/init.d`**

"Run level" directories are **`/etc/rc?.d`**
- Contain links back to **`init.d`** scripts
- Naming convention handles dependency order
- Remove/rename links to disable services

How do you actually prevent services from being started at boot time? Just so everybody's on the same page, let's quickly review the standard SYSV Unix boot scheme, so we understand how things evolved. What we're describing here is the traditional Unix `init`-based boot layout that was used up to RHEL6, Solaris 9, and in many other legacy Unix operating systems.

The traditional place for scripts that start/stop various services is `/etc/init.d` (though the directory name may change slightly from one Unix flavor to another, these are usually a link from `/etc/init.d` to the actual location for backwards-compatibility). Each script is supposed to control a single service, though Solaris, in particular, has historically been bad about lumping multiple services into a single script. The trick is ensuring that the services get started in the correct dependency order. For example, you wouldn't want to start your name server until the network interfaces on the system had been initialized, etc.

The system will typically have several `/etc/rc`<*n*>`.d` directories, where <*n*> corresponds to the "run level" the system is trying to boot to. Typically, run levels 3-5 correspond to normal "multi-user" Unix, with run levels 1-2 implying "single-user" mode or some other administrator-only access level. The default run level that the system boots to is defined in `/etc/inittab`—look for the line containing the string "`initdefault`". The `init` process will execute the scripts from the corresponding `/etc/rc`<*n*>`.d` directory.

Within the `/etc/rc?.d` directories are links back to the actual start-up scripts in `/etc/init.d`. The link names in the `/etc/rc?.d` directories are either "`S`*nn*…" (start this service) or "`K`*nn*…" (stop/kill this service), with the numbers after the `S`/`K` used to make sure that the scripts are executed in the correct order (typically you leave "gaps" to make it easier to insert new scripts).

If you want to prevent a service from being started at a particular run level, just remove the "`S`*nn*…" link from the appropriate `/etc/rc?.d` directory. Actually, you can just rename the link since the `init` process will completely ignore any script that doesn't start with a capital `S` or `K`.

There are lots of renaming conventions out there:

- Some sites will rename links "`000S`*nn*…" or "`zzzS`*nn*…" so that all the renamed links end up being listed at the beginning or end of the normal "`ls`" output.
- Other sites will use "`S`*nn*…" so that (depending on the default sorting algorithm used by your version of "`ls`") the original directory ordering is preserved as much as possible.
- I personally prefer "`.NOS`*nn*…" so that the renamed scripts won't appear unless you use "`ls -a`" to explicitly list the dot files.

The point is that renaming prevents the service from being started automatically, but re-enabling the service is as simple as restoring the original name. This is easier than removing the link entirely and having to remember where it used to sit in the dependency order when you want to restore the service.

Known Issues

- Serial script startup is slow
- Manually managing links is a pain
- Patches "put back" links

The traditional `init`-based boot system lasted for decades, but there were increasing complaints from users and administrators. One of the most obvious issues was that `init` simply runs scripts from the `rc?.d` directories in sequence. Typically, however, many services can be started in parallel. For example, once the network interfaces are initialized, you can start all of your other network-based services (web, email, time sync, etc.) at the same time because none of them depend on the others. This makes for much faster boot times.

The other issue was that creating and managing the shells scripts in `init.d` and the forest of links in the `rc?.d` directories was far from a great administrative interface. And vendor patch installs would often put the original links back into the `rc?.d` directories, effectively re-enabling the service. Admins had to re-audit their `rc?.d` directories after patch installs to make sure there were no unexpected changes.

Various solutions were created to address these issues. For example, RedHat borrowed a tool called `chkconfig` from SGI Irix that was a simpler command-line interface for managing the `rc?.d` directories. Changes made with this tool would not be clobbered by patch updates, etc.

Alternate boot systems were tried that could handle starting multiple services in parallel. Ubuntu introduced *Upstart* in 2006 as a replacement for the traditional `init`-based boot. Upstart was adopted by RedHat in RHEL6, though they kept the traditional looking `init.d` and `rc?.d` directory structure for backward-compatibility reasons. Solaris introduced the Service Management Framework (SMF) in Solaris 10 as a replacement for the traditional `init`-based boot sequence.

Linux `systemd`

- Parallel and "on demand" service startups
- Dependency "targets", not run levels
- Auto-restart failed services
- Patches don't overwrite local admin config
- No more shell scripting

While the Ubuntu project was working on Upstart, an alternative `init` replacement was being developed under the name `systemd`. To say that `systemd` was controversial in the Linux community is a huge understatement. The Wikipedia article (*http://en.wikipedia.org/wiki/Systemd#Adoption_and_reception*) does a good job of summarizing this controversy. Fedora adopted `systemd` as default in 2011 with Fedora 15, and smaller distros like Arch Linux and openSUSE followed in 2012. But `systemd` became a de facto standard when it was adopted by both RHEL7 (June 2014) and Debian 8 (April 2015). Surprisingly, Ubuntu announced that they would be following the Debian decision and adopt `systemd` in Ubuntu 15.04, effectively abandoning Upstart.

`systemd` is a very different animal from the traditional Unix `init`. Like Upstart, it's built around the idea of parallelizing the service start-ups as much as possible. But `systemd` introduces the idea of only starting services "on demand"—for example, only starting the SSH daemon the first time somebody tries to connect to port 22/tcp. Not only does this get the system up and running faster, it's also much better for multi-tenant virtualized environments because you don't get this huge load as every service on every guest operating system gets spun up at the same time.

`systemd` abandons the old concept of "run levels" and instead has various dependency "targets" that the boot process is aiming for. For example, there's a "networking" target that includes starting and configuring the network interfaces on the system, and a "multi-user" target that starts typical system services like SSH, web servers, etc. There's also a "graphical" target that starts the GUI-based login on the console. There's a dependency ordering so "networking" happens before "multi-user" which must be accomplished before "graphical" can go.

This dependency concept can also be extended to individual services. So, if you had a "stack" of services like a database, middleware, and a web server, you could arrange it so the web server would only start if the other two pieces were working. More interestingly, `systemd` can detect a failure in, say, the database service and gracefully shut down the other components. `systemd` can also be configured to automatically try and restart failed services.

Configuration for how to start and stop services has been moved to a simpler configuration file syntax, rather than requiring administrators and developers to write shell scripts to start and stop their services. For backwards compatibility, `systemd` does allow for `init`-style startup scripts, with a few minor incompatibility issues that are documented (see *http://www.freedesktop.org/wiki/Software/systemd/Incompatibilities/*).

`systemd` also introduces a consistent command-line tool called `systemctl` for managing services and even shutting down, rebooting, and hibernating systems. Changes you make with `systemctl` will be preserved across patch updates, etc. So, once you shut off a service with `systemctl`, it stays off until explicitly re-enabled.

systemd "Unit File" Example

```
[Unit]
Description=OpenSSH server daemon
After=network.target sshd-keygen.service
Wants=sshd-keygen.service

[Service]
EnvironmentFile=/etc/sysconfig/sshd
ExecStart=/usr/sbin/sshd -D $OPTIONS
ExecReload=/bin/kill -HUP $MAINPID
KillMode=process
Restart=on-failure
RestartSec=42s

[Install]
WantedBy=multi-user.target
```


Here's an example of a configuration file for a service started by `systemd`, in this case, the OpenSSH server. `systemd` refers to these configuration files as "unit files." The default unit files that ship with the operating system are found in `/usr/lib/systemd/system`. There's also a `/etc/systemd/system` directory where admins can put locally developed unit files, and even modified unit files, which should override the system defaults.

First, there's "`[Unit]`" declarations, which set basic parameters like the service "`Description`." You can see dependency information here, too—the OpenSSH server should only be started "`After`" the system accomplishes the `network` configuration target and the `ssh-keygen` service runs (the `ssh-keygen` service generates initial SSH host keys for the system if none exist).

In the "`[Service]`" description area are the parameters for starting ("`ExecStart`") and stopping ("`KillMode`" in this case) the service. `systemd` makes sure that all services are started in completely empty, sanitized environments. If your service needs options or environment variables set, you can up an external "`EnvironmentFile`" with the necessary settings.

OpenSSH (and many other Unix daemons) can be told to reread their config files without completely stopping and restarting the service; "`ExecReload`" provides the command for doing that. "`Restart=on-failure`" means that `systemd` will monitor the `sshd` process and start another one if the process dies. This is convenient for critical services like SSH; however, if you make a configuration error, you could make it impossible for `systemd` to start the server at all.

The "`[Install]`" section contains a hint that tells what target this service startup should be part of. This is used when the OpenSSH package is being installed on the system.

Admin CLI: `systemctl`

`list-units` – Give list of possible services

`enable`/`disable` – Start at boot?

`start`/`stop`/`restart` –
Manage currently running services

`status` – Detailed info and logs

`reboot`/`poweroff`/`hibernate`

Once the unit files have been created and placed in appropriate directories, the primary `systemd` administrative interface is the `systemctl` command. The basic syntax is "`systemctl <verb> <options>`." For example, "`systemctl list-units`" shows all active unit files (add "`--all`" to see both active and inactive units), while "`systemctl list-units -- type service`" shows only unit files for system services (as opposed to boot targets like "`network`," "`multi-user`," and other internal `systemd` unit files):

```
# systemctl list-units --type service
UNIT                    LOAD   ACTIVE SUB     DESCRIPTION
abrt-ccpp.service       loaded active exited  Install ABRT coredump hook
abrt-oops.service       loaded active running ABRT kernel log watcher
abrt-xorg.service       loaded active running ABRT Xorg log watcher
abrtd.service           loaded active running ABRT Automated Bug Reporting
…
```

The "`disable`" verb is how you stop a service from being started the next time the system boots—e.g. "`systemctl disable avahi-daemon.service`" to shut off the Linux daemon that handles the Apple Bonjour network protocol. "`systemctl enable httpd.service`" would cause Apache to be started the next time the system boots. You can also use "`start`" or "`stop`" to start/stop the service right now—"`systemctl start httpd`"—so you can begin testing your web configuration without waiting on a reboot. "`restart`" is just a "`stop`" followed by a "`start.`"

Speaking of reboots, "`systemctl reboot`" will reboot the machine. For all of us old school Unix types, the "`reboot`" and "`shutdown -r`" commands still work, too. "`systemctl poweroff`" shuts down the system completely (some of you may have used "`init 0`" for this on other Unix systems). There's even a "`sysctl hibernate`" for you Linux laptop users.

"`systemctl status …`" gives you a lot of detail about a given service:

```
# systemctl status sshd.service -l
sshd.service - OpenSSH server daemon
   Loaded: loaded (/usr/lib/systemd/system/sshd.service; enabled)
   Active: active (running) since Tue 2015-05-26 12:06:15 EDT; 2 days ago
 Main PID: 1420 (sshd)
   CGroup: /system.slice/sshd.service
           └─1420 /usr/sbin/sshd -D

May 26 12:06:15 LAB systemd[1]: Started OpenSSH server daemon.
May 26 12:06:15 LAB sshd[1420]: Server listening on 0.0.0.0 port 22.
May 26 12:06:15 LAB sshd[1420]: Server listening on :: port 22.
May 26 12:07:07 LAB sshd[2929]: Accepted password for sans from
192.168.176.1 port 39008 ssh2
May 26 13:07:25 LAB sshd[5905]: Accepted password for sans from
192.168.176.1 port 39153 ssh2
```

Up near the top of the output, you see basic information like what unit file was used to configure the service, the process ID and when the process was started, and the process state. The "`status`" output also includes logs from the service as well, which is pretty cool. Note the use of the "`-l`" flag (that's an "el" as in "long output"); without this option, the log lines are truncated at about 80 characters.

Note that technically you are allowed to leave off the "`.service`" for example, "`systemctl start httpd`." `systemd` assumes you're talking about a service unit if you leave off the extension. Still, as you level-up with `systemd` and start interacting with boot targets, startup ports, and sockets, etc., you will need to specify the extension for the non-service units. So, it's probably better not to get into bad habits. Make sure to type the "`.service`" each time.

Specific Recommendations

Many "standard" services can be disabled completely

Services may have ports for remote access that are not used:

- Disable ports when possible
- "Bind to loopback" trick is another option
- Some services require host-based firewall

At the beginning of this section, we talked about some general classes of services that can be disabled. In some cases, the decision as to whether or not to disable a given service is straightforward: "Is this a web server?", etc. However, there are some services where this is not so obvious and we'll talk about a couple of those services in more detail.

Obviously, you'll have to leave some services enabled, or you won't be able to do much with your machine. However, some of these active services (e.g., mail servers, NTP daemons, and sometimes older Syslog and those X Windows servers) enable network ports for remote access. In many cases, you don't need these ports and you should shut them down to reduce the potential for remote exploits. If you can't shut the port down, you might be able to configure the service to only listen on the loopback address. This is becoming a pretty standard configuration choice; recent Red Hat and Solaris default configurations use this tactic quite a bit.

If none of the above is an option, then you're stuck with using a host-based firewall to restrict access to these ports. We'll cover iptables in more detail later in today's course.

For now, let's look at some hooks for disabling some well-known, but largely unused services and service ports…

GUI Logins

- Login GUI (and underlying X server) may not be required if you never use the system console
- Change to "`multi-user.target`":

```
systemctl set-default multi-user.target
```

- Different strategies required for older Linux/Unix systems

If you happen to have installed the X Windows packages on your machine, then it's likely that the system is set up to automatically start the little X login GUI on your system's console. However, in many cases the console is never used and the system is only managed remotely via SSH. So perhaps the system doesn't need the graphical login, an X Windows server, and all of that underlying complexity.

On modern, `systemd`-based Linux systems, whether the GUI gets started or not depends on your default boot target. Most systems with X installed will boot to the "`graphical.target`" which starts the GUI at boot time. If you prefer to boot to a text console, simply change the default target to "`multi-user.target`" using `systemctl`:

```
# ls -l /etc/systemd/system/default.target
lrwxrwxrwx. 1 root root 36 May 14 05:50 /etc/systemd/system/default.target
-> /lib/systemd/system/graphical.target
# systemctl set-default multi-user.target
rm '/etc/systemd/system/default.target'
ln -s '/usr/lib/systemd/system/multi-user.target'
'/etc/systemd/system/default.target'
# ls -l /etc/systemd/system/default.target
lrwxrwxrwx. 1 root root 41 May 28 14:46 /etc/systemd/system/default.target
-> /usr/lib/systemd/system/multi-user.target
```

As you can see from the above output, the default target is just a symlink under `/etc/systemd/system`. You could manage this link by hand, but the `systemctl` command is less error-prone.

On older `init`-based Linux distros, whether or not to start the GUI was determined by the run level you booted to. Run level 5 started the GUI, while run level 3 meant multi-user mode without the GUI (similar to `systemd's "multi-user.target")`. The default run level was configured in `/etc/inittab`. Here's a sample `inittab` configuration line for booting the system to run level 3.

```
id:3:initdefault:
```

On other Unix platforms (e.g. Solaris), the graphical login is just another boot service that you can start or stop like a web server or mail server. Consult your vendor's documentation or the appropriate CIS Benchmark for information on which service to disable and how.

Legacy X Issues

- XDMCP
- X Font Service (**`xfs`**)
- X Remote Protocol (6000/tcp)

Note that if you do decide to leave this GUI enabled (on user desktop machines, for example), be aware that these GUIs generally support a legacy protocol called XDMCP (X Display Manager Control Protocol). This protocol allows remote devices like X terminals and Windows-based X desktop products to get a login widget like the one you would use to log into your local system. Very often, these X Windows logins *do not* log failed login attempts and can be an avenue for attackers to try exhaustive password guessing attacks, etc. While the modern Linux distros ship with the XDMCP port turned off, Solaris and many older Unix systems ship in a wide-open configuration; anybody on the network can request this login widget from your system.

You have the option of disabling XDMCP completely (on Solaris machines, copy `/usr/dt/config/Xconfig` to the `/etc/dt/config` directory and set "`Dtlogin.requestPort: 0`" in the new copy of the file). If you need to have XDMCP enabled in your environment, the `Xaccess` file can normally be used to only grant access to specific machines or IP addresses. You'll find this file in your X Windows configuration directory.

Legacy Linux and Unix systems may also start the X font server process, `xfs`. The `xfs` process listens on port 7100/tcp by default and typically runs as root. There have been remote exploits reported against various versions of this service. The service is not actually necessary since the X server can read the local font description files directly from OS directories. Disable this service. Some X server configuration file changes may be necessary for older Linux (RHEL4 and earlier) and Unix distros so that they can find the local font files.

If you run `netstat` or `lsof` on an older Unix system, you may find that your X server is listening on port 6000/tcp. This is the port used by the old, insecure X Remote protocol that allows you to display the graphical output of a process from a remote system on your local desktop. These days, we tunnel such traffic securely via SSH (more on this in a couple of days) and the old, clear channel 6000/tcp traffic is deprecated. You may be able to change the configuration of your `Xservers` file to start the X server with the "`-nolisten tcp`" option to stop the server listening on 6000/tcp. On very old systems, firewalling this port may be the only option.

Email Service

Elements of email delivery:

1. Send outgoing messages from local host
2. Handle queued messages
3. Listen on 25/tcp for *incoming* email

Only mail *servers* need to do #3

No 25/tcp means no external vulnerability

Every machine on the network is potentially capable of sending outgoing email to some other destination. For example, cron jobs automatically generate email from time to time and users on the system may use a mail client to compose and send email messages (this is Item #1 on the slide). These outgoing messages are generally transmitted immediately. However, if the message can't be sent immediately for some reason (say the network is down), it will be queued and the mail system will try to send the message later (Item #2).

Mail services will also listen on port 25/tcp for incoming messages from outside the system (Item #3). In this case, the machine is acting as a mail server receiving, forwarding, and storing email messages for your network. In typical environments, we don't use Unix systems for this anymore. Email is being handled by Windows mail services such as Exchange, or external cloud-based email providers like Gmail.

So, in theory, you can simply disable the 25/tcp listener on all of the Unix systems in your environment that are not acting as mail servers. It's possible that you could have 25/tcp shut down on *all* Unix machines because none of them act as mail servers. This would be a huge security win since it protects you from future potential email-based exploits and worms.

E-mail Issue

- Default config sends outgoing mail to `localhost:25`
- Disable 25/tcp, no outgoing email!
- Fix is to *only* listen on `localhost:25`

The problem is that the default configuration for Unix systems is to send outgoing email through the mail server at localhost:25. If you shut down the service listening on this port, then your outgoing email never gets sent!

On systems running Postfix as their default mail service (which includes Red Hat and many other Linux distros), the workaround is to run the listener on 25/tcp, but have it only bind to localhost and not accept connections from outside the system. That way, outgoing messages from the local system can be sent, but the mail service is protected from direct external access and compromise. Configure Postfix to only listen on localhost by setting `"inet_interfaces = localhost"` in `/etc/postfix/main.cf` (you will need to restart Postfix after making this change). Note that this is the default configuration on RHEL/CentOS.

On systems running Sendmail, set `DaemonPortOptions` setting in `/etc/mail/sendmail.cf` to force the daemon to bind only to the loopback address. However, with Sendmail, there is an alternate configuration option that sends outgoing email via an external server rather than localhost:25. This configuration allows you to shut off the Sendmail daemons completely, which is a somewhat cleaner way to live.

For further details, see the following articles:

`http://www.deer-run.com/~hal/sysadmin/sendmail.html`

`http://www.deer-run.com/~hal/sysadmin/sendmail2.html`

Other Legacy Issues

`inetd/xinetd`

SSH means these aren't needed

`syslogd`

Look out for listeners on 514/udp!

`inetd` (later `xinetd` on some Linux distros) was the traditional way of starting remote login (telnet, rlogin, rsh) and file transfer services (FTP) on Unix systems, as well as a number of other services. But the older, insecure cleartext login and file transfer protocols have all been replaced by SSH. So, it's possible you can run without `inetd`. In fact, most modern Linux distros do not even install `inetd` by default.

Older proprietary Unix systems such as Solaris may still ship with `inetd` enabled, however. In fact, there may be dependencies that make it unwise to shut off `inetd` on these systems. For example, the Solaris windowing environment requires `rpc.ttdbserverd`, which is started from `inetd`. Or you may have an enterprise backup utility that gets spawned from `inetd` (looking at you, NetBackup), requiring you to run `inetd` on even your Linux systems. The rule is to carefully audit your `inetd` configuration and remove all services that are not absolutely required. If none are required, then shut down `inetd` completely. If you do need to leave `inetd` running, you can usually turn on an additional "`inetd` tracing" feature so that the daemon logs every connection that is made to a service managed by `inetd`. Consult your vendor's documentation or the appropriate CIS guide for how to enable this.

Speaking of logging, traditional Unix Syslog servers listen on 514/udp for log messages from other systems. However, unless this machine is going to be a central Syslog server for collecting remote logs from other systems, it's best to make sure that your Syslog daemon isn't listening on 514/udp. After all, an attacker could barrage your `syslogd` with bogus messages to fill up your logging partition before launching an attack, or simply as a denial-of-service, or there could be an as yet unknown remote exploit in your Syslog daemon.

Modern Linux, BSD, Solaris, etc. systems generally do not listen on 514/udp by default. On older legacy Unix platforms, however, it may be necessary to explicitly configure `syslogd` not to listen on 514/udp. Again, consult the appropriate vendor documentation or CIS Benchmark guide.

Further Cleanup

Having stopped a service, remove its configuration files:

- **`inetd`/`xinetd`** config files and directories
- NFS, automounter config files
- Other server configs, server home dirs
- Useless crontab files

If you're not going to be using `inetd`/`xinetd`, it is safe to remove the `/etc/inet/inetd.conf` configuration file (note that `/etc/inetd.conf` is a link to `/etc/inet/inetd.conf` on Solaris) or `/etc/xinetd.conf` and `/etc/xinetd.d`. Similarly, we can get rid of NFS-related config files (Red Hat: `/etc/exports`, `/etc/auto.*`; Solaris: `/etc/auto_*`, `/etc/dfs/dfstab`;) and all of the other config files and directories used by the various servers we've disabled. Crontabs that are empty or contain only comments can also be purged.

The idea behind removing these files is to make auditing your system easier. If any of these files reappear on your system, you will know that something is **far wrong** with your machine. Also, removing the configuration file for a service will often prevent that service from starting normally if somebody (accidentally or on purpose) re-enables that service.

Lab Exercise

- Minimize your OS
- Get some practice with `systemd`

Exercises are in HTML files in two places: On the course USB media and in the home directory of the SANS user in the lab virtual machine. These two locations contain identical copies of the exercise files; it just depends on whether you prefer to look at them from a web browser inside the lab VM (because you're working in full-screen mode) or from your host operating system.

1. Open your web browser
2. Select "File… Open…" from the browser menu (Ctrl-O is the usual keyboard shortcut)
3. This is Day 1, Exercise 2, so navigate to *.../Exercises/Day_1/index.html* and open this HTML file
4. Click on the hyperlink that is the title of Exercise 2
5. Follow the instructions in the exercise

Session Hijacking and Sniffing

Before we talk about SSH, I want to talk about why using SSH is so important. So, we'll look at remote session hijacking exploits. In fact, I'll also be showing you a demo that you can run for the folks back at your company if they're being reluctant to migrate to SSH.

Session Hijacking

Attacker "takes over" session in progress:

- Attacker doesn't have to guess passwords
- May not be obvious to affected user

Two types of session hijacking attacks:

- Local attacks snoop streams via kernel
- Remote attacks from third-party machines

Session hijacking is an attack where the bad guy takes over a network session already in progress. If the attacker manages to take over your session after you've logged into a server and become root, then the attacker has suddenly achieved a root shell without ever guessing a single password! Naïve users may be completely unaware that this has happened, just writing the event off to a brief network outage that caused their session to lock up or be reset.

A local attacker with root privileges can snoop a network session in progress and/or enter extra commands via the appropriate tty devices and kernel structures. More insidious, however, is the ability for an attacker on some intermediate machine to take over a network session between two other computers. Session hijacking has been used in many famous attacks, including by Kevin Mitnick.

As far as exploit code goes, check out:

Tap – *ftp://ftp.cerias.purdue.edu/pub/tools/unix/sysutils/tap/*

Hunt – *https://packetstormsecurity.com/sniffers/hunt*

bettercap – *https://bettercap.org*

Tap does local session monitoring, and Hunt can be used to hijack telnet sessions in progress from some third-party intermediate machine. bettercap is another third-party session hijacking tool and has many cool features, including a suite of wireless attack tools. It should be noted that the possibility of session hijacking was first outlined by Steven Bellovin in his paper "*Security Problems in the TCP/IP Protocol Suite*" (ACM SIGCOMM #19, April 1989).

The Case for Encryption

- Encryption is only real defense against session hijacking
- Relying on switched network fabrics is no longer sufficient
- Still using cleartext protocols? Shame on you!

Any cleartext protocol is susceptible to session hijacking, not to mention having sensitive data snooped out of the transmission stream. Encryption not only protects the transmitted data but also makes session hijacking essentially impossible. In order to hijack an encrypted session, the attacker would have to determine the session key being used to encrypt the data between client and server. For a decent cryptosystem, this should be "computationally infeasible" (which is a fancy mathematical term for "darn near impossible").

Some sites still think they're secure because their network is entirely switched. They assume this prevents the attacker from seeing the data flowing between client and server, and thus preventing them from hijacking the session. Unfortunately, attackers have developed tools to force switches back into "hub mode" (traffic is mirrored on all ports just like an old non-switched hub) or to simply spoof MAC addresses to intercept traffic. Both `bettercap` (mentioned earlier) and `dsniff` (*http://monkey.org/~dugsong/dsniff/*) have this functionality built-in. Also, from a purely network snooping perspective, if the attacker compromises a critical machine like the company mail server, they can still see all the sensitive traffic flowing to and from that machine, even if the network is switched.

So, the bottom line is, if you're still using cleartext protocols on your networks (even for purely internal use), it's time to figure out a plan for getting away from them. Actually, that time came and went years ago, so it's time to get started!

SSH

This module looks primarily at the free OpenSSH distribution, which is the basis for the SSH software shipped with most Unix-like operating systems these days. However, most of the concepts covered here apply almost directly to the commercial SSH distributions provided by third-party vendors. This is because all of the available SSH distributions developed from the same original free distribution created by Tatu Ylonen.

This module is derived in part from original material created by Steve Acheson (`satch@employees.org`). Thanks, Steve!

What Is SSH?

An improved login/file transfer protocol:

- All sessions fully encrypted
- Supports strong authentication
- Available on a wide variety of platforms

Transparently tunnels X11 traffic–run remote GUIs securely!

Tunnels any TCP-based protocol–"quick-n-dirty" VPN!

As we discussed previously, administrators need to be able to log into their systems and transfer files back and forth between them in order to manage the machines. Administrators don't want passwords and other sensitive data to be stolen off the wire and want to avoid session-hijacking attacks. Encryption is the answer. SSH provides fully encrypted login and file transfer functionality, while at the same time providing (nearly) the same user interface as the standard `rlogin`/`rcp` commands.

Over time, however, SSH has acquired additional functionality. For example, SSH can automatically tunnel all X Windows traffic from clients via an SSH session to a remote machine back down the SSH tunnel to the local X server running on your system's display. This means that you can run a remote GUI application (like an administration front end) on the remote server and have all of the traffic securely encrypted and transported via an SSH tunnel. The best part is, you don't need to do anything special; usually, SSH takes care of all the details for you. In fact, it's easier to tunnel remote X events via SSH than it is to use the standard remote X protocol because SSH takes care of all of the authentication issues for you automatically! SSH can also tunnel other network protocols through an SSH login session as well, although you're limited to tunneling TCP-based protocols only. But this covers most useful internet protocols like HTTP, SMTP, POP, IMAP, etc. We'll look at SSH tunneling in more detail later in the course.

SSH is now ubiquitous. It's hard to find a Unix-like operating system that doesn't come with SSH already installed (this includes MacOS X). On Windows, you can get an SSH server if you're also willing to use the free Cygwin development environment (*www.cygwin.com*).

One of the historical stumbling blocks to enterprise deployment of SSH was the lack of Windows-like SSH clients, particularly file transfer clients with a standard "drag and drop" type interface. Happily, the

combination of PuTTY (*http://www.chiark.greenend.org.uk/~sgtatham/putty/*) and WinSCP (*http://winscp.net/*) seem to suffice for most users, and both are free software. Most Windows clients are now "fully functional" in the sense that they support X11 forwarding (if you have an X desktop running on your PC), port forwarding, etc.

If you would like a "drag and drop" type SFTP/SCP client for Mac OS, check out Fugu at *https://sourceforge.net/projects/fugussh/*

SSH "v1 vs. v2" Debate

SSH protocol v1 and v2 don't interoperate

SSH v2:

- Better design, more extensible, IETF standard
- Fixes protocol flaws in SSH v1
- Supports SFTP and warning banners
- Doesn't use patented ciphers
- *Slower than SSH v1*

Most sites are migrating to SSH v2 only

The question for most organizations is not whether or not to use SSH, but which version of the SSH protocol to employ. The original SSH protocol v1 does not interoperate with the new IETF draft spec v2 protocol, so generally v1 clients don't work with v2 servers and vice versa. OpenSSH servers are capable of handling both v1 and v2 clients if necessary, though they can also be configured to do one or the other exclusively (more on this later).

SSH protocol v2 is a ground-up rewrite of the v1 protocol. This rewrite was primarily motivated by two factors: (1) make the software easier to maintain, and (2) fix certain protocol problems in the v1 protocol (including some security fixes, but also things like better tunneling support, etc.). The v1 protocol has a reputation for being "insecure", but aside from a well-known man-in-the-middle attack, all of the security issues we know of in the v1 protocol have been fixed in some fashion. Protocol v1 does use some patented encryption algorithms (IDEA, for example) that may have licensing restrictions in some parts of the world.

The v2 protocol also includes additional functionality not found in v1. For example, v2 allows the display of statutory warning banners before login, which is not possible in v1 sessions. Protocol v2 is required for the SFTP file transfer protocol used by many "drag and drop" type file transfer clients. Also, protocol v2 was designed in a much more modular fashion that makes adding new functionality, new authentication types, and other features much easier.

The trade-off here, however, is performance. In part due to its modular nature, protocol v2 actually performs a little bit worse than protocol v1. Some sites are actually sticking with v1 for performance reasons. However, v1 has been officially deprecated and all future development efforts are focused on v2. For this reason, most sites are more comfortable going with v2 as a "standard."

`sshd` Command Line Options

- **`-f`** Specify alternate config file path
- **`-d`** Turn on debugging output (max verbosity is "**`-ddd`**")
- **`-i`** Daemon is running from **`inetd`**
- **`-D`** Do not "daemonize" (special boot management uses)

OpenSSH has a number of command-line options, but the most commonly used one is probably `-f` to specify an alternate configuration file path. Suppose, for example, that you installed OpenSSH from source into `/opt/openssh`. By default, the SSH server would look for its configuration file as `/opt/openssh/etc/sshd_config`, but you want to maintain this file in the `/etc/ssh` directory. Just use `-f /etc/ssh/sshd_config`.

The `-d` option turns on debugging. Adding additional `d`'s, up to a maximum of three, increases the verbosity of debugging.

As far as the other server command-line options, most of them can also be set as options in the server configuration file. My preference is to keep as much configuration information as possible in `sshd_config`. That way, I only have to go to one single file to see how my server is configured. If some options are set on the server command line and some in the config file, this gets to be a hassle.

However, some organizations will run redundant SSH servers via other mechanisms. For example, it's possible to run the SSH daemon from `inetd` with the `inetd`-spawned daemon to listening on an alternate port (use "`ssh -p` <*port*> <*hostname*>" to connect to a remote server on an alternate port). If the primary SSH server dies, you can always get in and fix the problem via the one spawned by `inetd`. Note that if you are running `sshd` from `inetd`, you *must* specify the `-i` option to `sshd` in the `inetd.conf` file.

Why not run the primary server from `inetd`? Well, each time `inetd` forks off a new `sshd`, the server will have to re-parse its configuration file, generate keys, etc. This adds a lot of latency to new connections. This is why most sites prefer running SSH as a stand-alone daemon.

On older `init`-based systems, a trick was to invoke the SSH daemon at boot time directly out of `/etc/inittab`, rather than via a regular boot script. The advantage to this approach was that you could use the "`respawn`" option in `/etc/inittab`, which tells `init` to automatically restart the SSH daemon if it dies. The `init` program effectively becomes a "watchdog" for the SSH daemon.

The only caveat here is that you had to prevent the SSH daemon from doing the normal "daemonizing" technique of spawning a child process and killing the parent. The `init` program would be fooled into thinking that the SSH daemon had died and would constantly be trying to fire off new copies of the daemon. This would be bad. The `-D` option tells the SSH daemon not to "daemonize", so always use this option when running from `init`. This option is also used in the standard `systemd` SSH startup on modern Linux distros for similar reasons.

Basic OpenSSH Config Options

`Port` (default: 22)

`ListenAddress` (default: 0.0.0.0)

IP addr and port to listen on; must specify **`Port`** *first*

`Protocol` (default: 2,1)

Specify "2" for SSH v2 server only (recommended)

`MaxAuthTries` (default: 6)

Number of login attempts permitted before session terminates; failure logging starts at **`MaxAuthTries/2`**

`Banner` (default: none)

Path name of file containing banner text

Now let's turn our attention to the server configuration file, `sshd_config`. By the way, even though the config file options generally use a specific capitalization scheme, the option names are, in fact, case insensitive.

`Port` and `ListenAddress` are used to specify the port and IP address to listen on. The default is to bind to port 22 (TCP) on all interfaces on the system. But if you wanted to bind to an alternate port or specific IP address, you can. For some reason, `Port` must be specified in the configuration file *before* `ListenAddress`. This is the only case that I know of where the order of configuration items in the file is significant (and I can't see why it really matters since the server has to parse the whole darn config file anyway).

`Protocol` defines which version of the SSH protocol the server should support. The default is "2,1" which means to support both as requested by the client. Generally, I recommend forcing people to migrate to the v2 protocol exclusively. So "2" is the right choice in my book unless you have to support v1 clients for application compatibility reasons. Note that commercial third-party SSH distributions may not support this parameter as they have decided to support protocol v2 only.

`MaxAuthTries` is the number of login attempts the remote user gets before the server dumps the connection and forces the user to reconnect. Note that the server doesn't start complaining about failed login attempts until half of the allowed attempts have failed (actually, the version of SSH that comes with Solaris implements another parameter, `MaxAuthTriesLog`, to control logging and I keep hoping this makes it into the main OpenSSH distribution). The default for this parameter is pretty large; I often tune this value down to "3".

`Banner` is the path of a text file containing whatever statutory warning message you want to display prior to login. As noted previously, this option only works on v2 sessions.

Options to Fix Simple Problems

SyslogFacility (default: AUTH)
LogLevel (default: INFO)

Specifies Syslog facility and how much to log

StrictModes (default: yes)

Ensures user files/homedirs are not world-writable

PrintMotd (default: yes)

Set to "no" if you're seeing the motd twice

PIDFile (default: **/var/run/sshd.pid**)
XAuthLocation (default: varies)

Can be used to customize paths as necessary

The SSH daemon logs to the Syslog `AUTH` facility by default, and this is almost always the correct choice. Linux systems use `AUTHPRIV` instead. `LogLevel` controls the verbosity of logging. Again, the default here is probably what you want, but you may want to enable higher levels of logging when you're trying to debug a problem. See the documentation about this option in the `sshd_config` manual page.

`StrictModes` defines whether or not the SSH daemon will check the permissions on the user's home directory, `~/.ssh` directory, and the files in the `~/.ssh` directory and make sure they don't have unsafe write permissions. The default is to do the checks, and generally, SSH knows what it's talking about. However, if there's something pathological at your site where you just can't run with `StrictModes`, you can disable it, but I urge you to try fixing your environment first before disabling `StrictModes`.

`PrintMotd` controls whether the SSH daemon displays the `/etc/motd` file before starting the login shell. On some systems, the login shell or the system PAM configuration does this as well and you end up seeing the `motd` file twice. Just set `PrintMotd` to "no" and this problem goes away.

On some systems, the `xauth` program (required for X tunneling) may be in a non-standard location. You can customize this path name with the `XAuthLocation` directive. Similarly, you may want to put your `sshd.pid` file in a non-standard location–customize this with the `PIDFile` directive. Usually, however, the defaults for these pathnames that are compiled into the SSH daemon are appropriate.

Not Quite Cut and Dried…

`AllowTCPForwarding` (default: yes)

`X11Forwarding` (default: no)

Note that users can still run their own servers on high ports if they don't like your settings

`PermitRootLogin` (default: yes)

*"**`no`**" disables direct root logins (this should be the default)*

*"**`without-password`**" stops logins w/ passwords (keys still work)*

*"**`forced-commands-only`**" requires key with forced command option set (described later)*

Some sites are concerned about the security implications of allowing TCP port forwarding or X11 session forwarding. The concern is that an attacker with access to one end of the tunnel might be able to compromise the system on the other end of the tunnel. You can disable this functionality as shown here (though frankly, it's really useful as we'll see later in the curriculum), but you should be aware that truly motivated users can just fire up their own SSH servers on high ports and allow forwarding.

`PermitRootLogin` controls whether or not users are allowed to log into the system via SSH directly as the root user. As we'll discuss elsewhere, these kinds of "anonymous" root logins are not a good idea, so setting this to "yes" is never the right thing. Unfortunately, "yes" is the default and most vendors ship their SSH daemon in the default setting. *sigh*

It turns out that there are other settings for this option besides "yes" and "no." The "`without-password`" choice looks alarming, but what it really means is that direct root logins are allowed as long as you're using something better than regular password authentication. This is useful because it means that you can set up public key style authentication for automated `cron` jobs (I'll show you an example later in the course) but still prevent normal users from logging in as root.

The "`forced-commands-only`" choice is even more restrictive. This setting means that root logins are only allowed via public key authentication, and even then, only if the public key has a specific "forced" command associated with it. More on this when we talk about the format of the `authorized_keys` file later in the course.

Privilege Separation

- Many OpenSSH exploits due to authentication libs
- PrivSep runs authentication in **`chroot()`**ed environment
- Requires special user/group and simple directory setup
- Some early functionality issues—should be OK now

If you look at most of the remote SSH exploits, the vulnerabilities have occurred in the very early stages of the connection process. In particular, a goodly number of the attacks have been due to vulnerabilities in things like the OpenSSL encryption libraries or the Kerberos and S/Key libraries used during authentication, rather than in the SSH code base per se. Hence, these have been vulnerabilities that are largely out of the control of the SSH developers.

Privilege separation (aka "privsep") is an attempt to mitigate future vulnerabilities of this type. The idea is that the privileged SSH process that's handling the new session `fork()`s a child process that immediately `chroot()`s itself to an empty directory and drops privileges. This unprivileged child process handles the session negotiation and user authentication (any privileged information the child needs is obtained from its parent via a rigidly controlled interface). This way, if there are any vulnerabilities in these early stages, the attacker has a much more difficult time actually getting control of the system. Once the session negotiation and authentication have been completed, the unprivileged child process passes its state information back to the parent and then exits. The parent then `fork()`s another child process that immediately takes on the user's privileges and starts the user's shell.

For privsep to work, you need to create an unprivileged user and group (both called `sshd` by default—use a unique UID/GID) and an empty directory (`/var/empty` by default—should be mode 111 owned by root:root) for the `chroot()`. There have been issues with privsep on various platforms. See `README.privsep` in the OpenSSH source distribution for more information. Generally, privsep is an important security feature. You should try running with privsep enabled by default, and only turn it off if you're having problems.

Authentication/User Options

PasswordAuthentication (default: yes)

PermitEmptyPasswords (default: no)

Whether or not to allow standard passwords
You should never allow empty (null) passwords

PubkeyAuthentication (default: yes)

Allow public-key based authentication? (Protocol v2)

AllowUsers/DenyUsers

AllowGroups/DenyGroups

Comma-separated list of users/groups to allow/deny
Can use wildcards here (, ?)*

`PasswordAuthentication` controls whether you want to allow users to log in with their normal Unix password. If you enable this (and most sites do), then always make sure that `PermitEmptyPasswords` is set to "no", so that users aren't allowed to log in with null passwords. Of course, you shouldn't allow users with null password fields in `/etc/shadow`, but this is an additional layer of defense.

`PubkeyAuthentication` allows users to log in via public keys in their `authorized_keys` file. We'll talk more about public key authentication on Day 3. Most sites allow this form of authentication. Note that if your SSH daemon also supports v1 sessions, then you also want to turn on `RSAAuthentication`, which is the v1 equivalent.

Even though a user has an account on the system, you can still block them off the machine with the `AllowUsers`/`DenyUsers` options. These options take comma-separated lists of usernames (numeric UIDs are *not* allowed) to allow or deny. The syntax "*user@host*" is also supported to block users only from a specific host. You can also use wildcards (* and ?). `AllowGroups`/`DenyGroups` is the same, but it applies to group names that the user might be a member of.

By the way, the order of precedence here is:

1. Check DenyUsers, reject connection if username is listed
2. Check AllowUsers (if set), only allow connection if user is listed
3. Check DenyGroups, reject if user is member of any denied group
4. Check AllowGroups (if set), only allow users in one of the listed groups

This behavior is not documented. I had to read the source code…

UsePAM and You

- As of OpenSSH v3.7 "**`UsePAM yes`**" tells server to obey local PAM config
- Effectively overrides **`PasswordAuthentication`** option
- Bug with **`PermitRootLogin`** in 3.x releases

Newer versions of SSH have a `UsePAM` setting that tells SSH to obey the standard PAM (Pluggable Authentication Modules) configuration in your operating system. This is convenient if you use LDAP authentication or some other third-party authentication system that is configured via your PAM stack. For example, here's a how-to guide for hacking Google two-factor authentication into SSH via PAM, *http://xmodulo.com/two-factor-authentication-ssh-login-linux.html*. PAM is also how you typically hook into OS-specific functionality like password expiration, etc.

In general, it is recommended to set "`UsePAM yes`" in the `sshd_config file`. However, this setting basically overrides the `PasswordAuthentication` option, since your PAM configuration might allow normal Unix password logins, even if you set "`PasswordAuthentication no.`"

In the OpenSSH 3.x releases where UsePAM was introduced, there was some real strangeness with the `PermitRootLogin` option when UsePAM was enabled. Specifically, if you set "`UsePAM yes`" and "`PermitRootLogin without-password`", PAM would get in the way and may allow the user to authenticate using the root password. Basically, PAM would be telling the SSH daemon that the user has presented a valid credential, and since the SSH daemon was configured to allow root logins with a valid credential, the user would get logged in with just a password and without having the right private root key. This is not what you want!

This issue has been fixed since OpenSSH 4.0 (circa 2005). Upgrade.

```
Port 22
ListenAddress 0.0.0.0
Protocol 2
SyslogFacility AUTH
LogLevel INFO
Banner /etc/issue
PidFile /var/run/sshd.pid

HostKey /etc/ssh/ssh_host_rsa_key
HostKey /etc/ssh/ssh_host_dsa_key
# HostKey for protocol version 1
#HostKey /etc/ssh/ssh_host_key
# Lifetime and size of ephemeral version 1 server key
#KeyRegenerationInterval 1h
#ServerKeyBits 768

UsePrivilegeSeparation yes
UsePAM no
StrictModes yes
MaxAuthTries 3
LoginGraceTime 2m
PrintMotd no

PasswordAuthentication yes
PermitEmptyPasswords no
PermitRootLogin no

# Enable this for v1 compatibility
#RSAAuthentication yes
PubkeyAuthentication yes

# Also for v1 servers
#RhostsRSAAuthentication no
HostbasedAuthentication no
IgnoreUserKnownHosts yes
IgnoreRhosts yes

# Change to "yes" to enable s/key passwords
ChallengeResponseAuthentication no

AllowTcpForwarding yes
X11Forwarding yes
GatewayPorts no

# Make sure path to executable is correct below
Subsystem   sftp    /usr/local/libexec/sftp-server
```

The Canonical Question (1)

Can I disable the host key prompt on new connections?

- "Correct" solution: Maintain the global **`known_hosts`** file
- Client can now validate public keys automatically
- Unfortunately, this adds an extra administrative burden

Every time one of your users logs into a new machine, they see:

```
The authenticity of host 'foo (192.168.1.1)' can't be establ…
RSA key fingerprint is 09:ad:55:2e:39:f0:78:e8:61:ac:ff:9d:7a:…
Are you sure you want to continue connecting (yes/no)?
```

The user always types "yes", although what they're actually supposed to do is type "yes" and then verify the key fingerprint once they get into the remote system using the "`ssh-keygen -l`" command (if they know where to find the host key for the system—most users don't). This process really needs to be scripted better so that normal users have a hope of actually validating keys successfully.

Users generally want you to make this prompt "go away." One way to accomplish this is to maintain the "global `known_hosts` file" of the public keys of all systems on your network. If this file exists on the *client* system, then the client can validate the public key automatically. The user won't be prompted.

However, gathering and maintaining a file of public keys for all hosts on your network is an administrative burden. Remember that host keys tend to change every time you reinstall a system. OpenSSH comes with an ssh-keyscan program, which can help gather keys from the network, but maintaining the file is still a hassle.

The Canonical Question (2)

Lazy solution via a *client* configuration (not recommended):

`StrictHostKeyChecking` (default: ask)

*"**`no`**" disables prompt (all keys accepted automatically).*
*"**`yes`**" prevents connection unless key already in **`known_hosts`***

Note—can't enforce strict default due to client overrides

The other alternative, which I strongly suggest you *avoid,* is to set "`StrictHostKeyChecking no`" in the default client configuration on your systems. This setting basically tells the client to just blindly accept all public keys without prompting the user, making your users more vulnerable to man-in-the-middle attacks. It's worth noting that even if you make this setting, users will still get the warning message when the public key for the remote system changes, so it's not entirely blinding the user to potential problems. By the way, the setting "`StrictHostKeyChecking yes`" means that the client will *never* accept unknown public keys. If the machine the user is connecting to isn't listed in the global `known_hosts` file, then the user is not allowed to connect.

These settings are made in the global client config file. Typically, this file is `$INST/etc/ssh_config` (note that the filename differs from the server config filename only very slightly—it's "`ssh_config`" rather than "`ssh`***`d`***`_config`"). Settings that should apply to all client connections appear after the "`Host *`" line:

```
Host *
StrictHostKeyChecking yes
```

Note that the user can override this setting either in the client config file in their home directory (`~/.ssh/config`) or on the SSH client command line, so there's no way for the administrator to absolutely enforce this setting for all user sessions.

Host-Based Firewalls

We've talked about minimizing your systems' "external vulnerability profile" by disabling services, etc. Host-based firewalls can be used to restrict what network traffic the system will accept, further reducing the number of entry points for external attackers. As network-layer perimeter security mechanisms become increasingly porous to attack, implementing filtering on individual systems becomes more and more important.

Filter Network Access

- Use network firewalls to enforce general policy and drop obviously bogus traffic
- Use host-based firewalls as an additional layer and for specific applications
- Don't forget access controls built into the application itself
- RPC-based services can be troublesome, not appropriate for use *through* firewalls

Generally, some combination of both network layer filtering at your firewalls and other gateways along with host-based filtering is appropriate. Network layer filtering not only is the first line of defense against external attackers but is also the place where you prevent obviously bogus traffic from reaching your hosts. For example, spoofed traffic (traffic from "outside" that is sourced from an internal IP address) can be dropped, along with traffic from reserved address spaces (the loopback network 127.0.0.0/8 and the RFC 1918 network space). Source-routed traffic, maliciously fragmented traffic, and directed broadcasts can also be shut down. Host-based filters like iptables can be used on individual hosts for additional security, and of course, many applications have this sort of filtering built-in. This is becoming increasingly important as the network perimeter becomes more and more porous.

Note that RPC-based services tend to be very difficult for firewall administrators. Each time an RPC service starts up, it gets a new ephemeral port number, making it essentially impossible to write a rule to safely allow access to the service via a firewall. Still, this is not generally a huge deal since the relatively insecure authentication implemented by RPC-based services make them something that you only want to run *behind* a layered firewall defense and not expose to the outside world.

Many Linux distributions have IP-address-based filtering functionality built into their portmapper and other RPC daemons. Check the system manual pages for more information.

Down into the Weeds

Going to use **`iptables`** command-line:

- Better understanding of what's happening
- Easier to automate

Frankly, command-line syntax is a little confusing at first

Many different firewall administration tools listed in notes

I'm the first person to admit that the Linux `iptables` command-line interface is a little clunky and confusing, particularly when you're seeing it for the first time. But I want to show you how to create firewalls using this interface for a couple of reasons. First, I want you to understand exactly what's happening in terms of firewall rules so that you can "translate" the lessons learned here to the host-based firewall syntax for your particular flavor of Unix. Second, if you're going to automate firewall deployment across a network of Linux machines, you're likely to run into the `iptables` CLI at some point.

Lots of people obviously don't like the `iptables` CLI, because there is a plethora of different firewall management options available. Red Hat-based distros are now using a management layer on top of iptables called firewalld (*https://fedoraproject.org/wiki/FirewallD*). Ubuntu ships a simplified command-line interface called UFW, which is actually pretty decent. Other Open Source tools include Firestarter (*http://www.fs-security.com/*) and Firewall Builder (*http://fwbuilder.sourceforge.net/*). Firewall Builder actually lets you manage a network of heterogenous firewalls, including iptables, pf (BSD), and Cisco.

There are even bootable Linux distros that you can use to turn an off-the-shelf PC into a network firewall device: See Smoothwall (*http://smoothwall.org/*) and Untangle (*http://www.untangle.com/*). pfSense (*http://www.pfsense.org/*) is a BSD-based project that's similar.

Preparing the Ground

- Flush any pre-existing rules
- Apply "default deny" policy

```
iptables -F

iptables -P INPUT DROP
iptables -P OUTPUT DROP
iptables -P FORWARD DROP
```


OK, let's begin by clearing any existing rules using "`iptables -F`" ("`-F`" for "flush"). By the way, you can list the current rules sets using "`iptables -L`" ("`-L`" for "list"). Personally, I always use "`iptables -L -v`", which gives me packet and byte counts filtered by each rule. You can use "`iptables -Z`" to zero these counters at any time.

Next, we want to put our firewall into the typical "default deny" stance. You do this by setting a policy ("`-P`") of `DROP`. The tricky bit is that iptables has different rule sets—usually referred to as *chains* for historical reasons, for inbound and outbound traffic. So, we need to set our default policy for both the `INPUT` and the `OUTPUT` chains.

There's also a `FORWARD` chain that applies to packets being routed through the system. This is useful when you're trying to turn a Linux system into a network-layer firewall, but not useful to us when creating a host-based firewall. We'll just set a default `DROP` policy here and then forget about the `FORWARD` chain for the rest of this example.

By the way, the `DROP` action causes the firewall to drop packets silently. If you prefer to send something back to the originating host when you drop a packet, you can use the `REJECT` keyword instead. The default response from the `REJECT` action is ICMP "port unreachable." Read about the "`--reject-with`" option in the `iptables` manual page if you're interested in customizing the type of response packet.

Don't Mess with Loopback

- Allow traffic freely on loopback interface
- All other net 127.0.0.0 traffic is spoofed

```
iptables -A INPUT -i lo -j ACCEPT
iptables -A OUTPUT -o lo -j ACCEPT

iptables -A INPUT -s 127.0.0.0/8 -j DROP
```


Great. At this point, we've now got a "default deny" policy and a completely empty rule set. That means we're currently dropping all network activity in and out of the system. Guess we'd better do something to fix that, huh?

Generally, the first rules you add to any firewall policy are rules to leave all loopback traffic unmolested. Loopback traffic is generated between processes within this machine and is both normally safe and typically critical to the operation of the system. So, we're going to append ("`-A`") rules to both the `INPUT` and `OUTPUT` chains to `ACCEPT` traffic on interface `lo`. Notice that it's "`-i lo`" on the `INPUT` rule because we're specifying the "input interface", but it's "`-o lo`" on the `OUTPUT` rule to specify the "output interface".

The loopback interface is the only place you should be seeing net 127.0.0.0/8 traffic. So, once we've created rules to pass the loopback traffic, we can add a rule to drop any other traffic trying to reach the box that purports to come from this network ("`-s`" to specify source address). This is an example of what's usually referred to as an "anti-spoofing" filter.

This filter also demonstrates another important aspect of iptables firewall rules. They are "first match and exit" type rules—meaning that once a rule matches a given packet, then the action specified with "`-j`" takes effect immediately and no further rules are processed. It's important that the first two rules on the slide above take care of all loopback traffic before the third rule, which would terminate all such traffic because it will be using net 127.0.0.0/8, can be applied.

By the way, while we'll be using the "`-A`" flag to append rules to chains throughout this section. It's worth noting that there's also an "insert" option, "`-I`", that allows you to insert rules at the front or someplace in the middle of a chain. While this is useful in emergency situations, I'm actually going to recommend a different strategy for managing your firewall rule sets that make "`-I`" less useful. More on this in a moment.

Allow Our Packets Out…

- Enable stateful session handling
- Could do egress filtering if desired

```
iptables -A OUTPUT -p tcp -m state \
   --state NEW,ESTABLISHED -j ACCEPT
iptables -A OUTPUT -p udp -m state \
   --state NEW,ESTABLISHED -j ACCEPT
iptables -A OUTPUT -p icmp -m state \
   --state NEW,ESTABLISHED -j ACCEPT
```


Host-based firewalls, particularly on personal machines like laptops and desktop workstations, tend to be permissive in the outbound direction. Here, we're allowing all outbound TCP, UDP, and ICMP traffic.

Iptables does allow you to filter traffic based on destination IP address ("`-d`") and destination port ("`--dport`") and even ICMP message type ("`--icmp-type`"). So, you could create more restrictive rules to only allow certain types of outbound traffic (egress filters). Most people don't bother, in my experience.

Notice the "`-m state --state NEW,ESTABLISHED`" options in each of our rules. Iptables is a "stateful" firewall if you enable "`-m state.`" Here, we're allowing outgoing packets that either initiate `NEW` connections or which are part of `ESTABLISHED` connections. The kernel tracks the `NEW` network connections that we initiate and can automatically allow the return packets back in if we put the appropriate hooks in our `INPUT` chain…

And Allow Return Packets

- Only allow inbound packets from sessions we've initiated

```
iptables -A INPUT -p tcp -m state \
   --state ESTABLISHED -j ACCEPT
iptables -A INPUT -p udp -m state \
   --state ESTABLISHED -j ACCEPT
iptables -A INPUT -p icmp -m state \
   --state ESTABLISHED -j ACCEPT
```


All we need to do on the `INPUT` side is to allow the inbound packets that are part of `ESTABLISHED` connections that we initiated. We're not allowing any `NEW` connections inbound, so as of right now nobody can access any of the networked services on our system from outside.

What About External Access?

- Can selectively enable access by port
- Personal machines may not need this

```
iptables -A INPUT -p tcp --dport 22 \
   -m state --state NEW -j ACCEPT
iptables -A INPUT -p tcp --dport 80 \
   -m state --state NEW -j ACCEPT
```


However, maybe you want people to be able to access services on this machine, like SSH for remote administration, or port 80/tcp because the machine is a web server. Feel free to "drill holes" in the firewall policy for the system by granting access to specific ports as needed. On my own personal laptop, I don't tend to do this because there's no reason for anybody to be accessing the machine from outside. Obviously, on machines that are acting as servers, you will need to allow some external access.

Here we're just allowing `NEW` connections on the specified ports. Remember that we already have generic rules on the `INPUT` filter that allow packets from `ESTABLISHED` connections.

What About Logging?

- Insert non-terminal `LOG` rules anywhere
- Let's finish chains with `LOG` rules to catch packets before default drop policy

```
iptables -A INPUT -j LOG
iptables -A OUTPUT -j LOG
iptables -A FORWARD -j LOG
```


Typically, you'd like to get some logging from your firewall. Different sites have different philosophies on this. Personally, I don't care about traffic that's being permitted by my firewall (I wouldn't have permitted it if it weren't OK), but I am curious about traffic that's being dropped. Dropped traffic could alert you to attackers trying to get into your box, or possibly to mistakes in your firewall policy that are blocking legitimate traffic.

Here, we are appending one final rule to our entire rule set chains that will `LOG` any traffic that makes it through all of the other rules. Unlike all other iptables rules, `LOG` rules are "non-terminal", which means that other rules will be processed even after the packet matches a `LOG` rule. This means you can insert `LOG` rules in the middle of your chains to `LOG` whatever traffic you want.

Iptables logs end up being spewed out of the kernel, picked up by `klogd`, and ultimately deposited in your `/var/log/messages` file. Or you can configure Syslog to dump them someplace else if you want. On many Linux systems, they're also sent to the console (which can be a huge hassle if you're actually trying to use the console).

Dealing with `iptables`

- If you make piecemeal changes to rules, you end up with unmaintainable mess
- Better to create a script to apply rules, so you have documented baseline
- Make updates to script and execute to update filtering rules

You could sit on the console of the system entering all of these `iptables` commands by hand. The risk, however, is that you make a typo, or forget to add a rule or make some other trivial error that causes your firewall to fail. A much better approach would be to create a script file with all of your `iptables` commands. When you want to make a change, all you have to do is tweak the script and then rerun it. You can also run the same script on multiple systems to have a consistent policy. Finally, the script itself documents your firewall policy without having to dump it out via "`iptables -L`".

Saving Your rule sets

- Current filtering rules not saved across reboots by default
- Dump rules with `iptables-save`
- Add this to the end of `iptables` configuration script???

When you create rules with the `iptables` command-line interface, those rules will not be preserved by default when the system reboots. The `iptables-save` program will dump the current rule sets in a form that can be read in by the `iptables-restore` program at boot time. The typical Linux convention is that your stored rules live in `/etc/sysconfig/iptables`, so you typically redirect the output of `iptables-save` into this file: "`iptables-save >/etc/sysconfig/iptables`".

Note that on Red Hat 6.x and earlier, running "`/etc/init.d/iptables save`" would do this automatically. But since Red Hat 7.x moved to `systemd`, this option is not available. Red Hat 7.x systems are now using Firewalld for managing iptables, and the save functionality has migrated there.

If you're following my suggestion from the last slide and keeping all your `iptables` commands in a script file, then you may want to consider adding a call to `iptables-save` at the end of the script so that it happens automatically. Some sites prefer not to do this, just to avoid the chance of overwriting their "known good" rule set with something experimental and broken. Some sites actually put a call to `iptables-save` in a script in their boot directories so that the current firewall state is saved as the system shuts down. Personally, I think this is dangerous. What if you're rebooting the machine because your firewall is hosed due to an improper update?

Pathological Protocols

- "Active" FTP makes connections *back*
- Your firewall is configured to block this
- Trick: Allow `RELATED` packets inbound

```
iptables -A INPUT -p tcp -m state \
   --state ESTABLISHED,RELATED -j ACCEPT
iptables -A INPUT -p udp -m state \
   --state ESTABLISHED,RELATED -j ACCEPT
iptables -A INPUT -p icmp -m state \
   --state ESTABLISHED,RELATED -j ACCEPT
```


The rules we've implemented so far actually work fine for most systems. One thing you might notice, however, is that some FTP clients on the machine may have trouble getting remote directory listings or transferring files to and from remote systems. That's because the standard FTP protocol actually calls for the remote server to initiate network connections back to your client machine whenever you want to get a directory listing or transfer a file. Your firewall is configured to block new connections coming back into your system, so these attempts fail.

One workaround is to use "passive" FTP for everything (you'll notice the Linux FTP clients are generally configured to use "passive" FTP by default). With "passive" FTP, the client initiates all connections, not the server. However, you sometimes run into problems where the FTP server you're talking to is sitting behind a firewall that prevents you from making the passive FTP data connections.

The better solution is to use an additional feature of the iptables state mechanism and allow packets that are `RELATED` to sessions currently active on the system. This covers not only traffic like the ephemeral FTP data connections, but also traffic like ICMP administrative messages ("packet too big", "port/host unreachable") that you might also want to receive. So, it's probably a good idea to update your `INPUT` chains to allow both `ESTABLISHED` and `RELATED` traffic.

FTP Still Doesn't Work!

- Have to load app-specific kernel module
- Needs to happen automatically when the system reboots
- See notes for configuration details

The problem is that updating your rules to allow `RELATED` traffic is not sufficient to get FTP working. You actually also have to load a special kernel module that can monitor your FTP sessions, recognize when an ephemeral data connection is expected, and pass this traffic appropriately. These kinds of filters are necessarily application-specific because they have to monitor the commands going back and forth between client and server and take action. Linux distros include modules not only for FTP but also other "pathological" protocols like H.323.

For Red Hat 7.x, the modules are found at `/usr/lib/modules/*/kernel/net/netfilter/nf_conntrack_*.ko` – the FTP module is `nf_conntrack_ftp.ko`. To automatically install this module at boot time, edit `/etc/sysconfig/iptables-config` and add `nf_conntrack_ftp` (without the "`.ko`") to the `IPTABLES_MODULES` configuration.

For kernel version 2.6.19 and earlier, aka Red Hat 6.x and older, the modules are found at `/lib/modules/*/kernel/net/ipv4/netfilter/ip_conntrack_*.ko`. Note the change in module names—the FTP module is `ip_conntrack_ftp.ko` in these versions. You load the module automatically at boot time by adding the following line to `/etc/modprobe.conf` (this configuration *must be entered as a single long line* in `modprobe.conf` or your system may not boot, be careful!):

```
install ip_conntrack /sbin/modprobe --ignore-install ip_conntrack;
/sbin/modprobe ip_conntrack_ftp
```

The configuration entry tells the system that when the `ip_conntrack` module is loaded (this module is part of the iptables state mechanism), also load the `ip_conntrack_ftp` module.

```
#!/bin/sh
# Sample iptables configuration script

PATH=/sbin
export $PATH

# Flush previous rules
iptables -F

# Set "default deny" policy
iptables -P INPUT DROP
iptables -P OUTPUT DROP
iptables -P FORWARD DROP

# Clear traffic on loopback interface
# All other network 127 traffic should be dropped
iptables -A INPUT -i lo -j ACCEPT
iptables -A OUTPUT -o lo -j ACCEPT
iptables -A INPUT -s 127.0.0.0/8 -j DROP

# Allow inbound SSH connections
iptables -A INPUT -p tcp --dport 22 -m state --state NEW -j ACCEPT

# Allow other inbound traffic that's part of connections we've started
iptables -A INPUT -p tcp -m state --state ESTABLISHED,RELATED -j ACCEPT
iptables -A INPUT -p udp -m state --state ESTABLISHED,RELATED -j ACCEPT
iptables -A INPUT -p icmp -m state --state ESTABLISHED,RELATED -j ACCEPT

# Allow all outbound traffic
iptables -A OUTPUT -p tcp -m state --state NEW,ESTABLISHED -j ACCEPT
iptables -A OUTPUT -p udp -m state --state NEW,ESTABLISHED -j ACCEPT
iptables -A OUTPUT -p icmp -m state --state NEW,ESTABLISHED -j ACCEPT

# Log any other traffic before it gets whacked by default policy
iptables -A INPUT -j LOG
iptables -A OUTPUT -j LOG
iptables -A FORWARD -j LOG
```

Enough!

This is enough to get you going…

Lots of other functionality available:

- Connection rate-limiting
- NAT, in-line packet mangling
- Passive OS fingerprinting

Plenty of How-Tos available on the internet

OK, you've now got a basic, but fully functional host-based firewall policy for your systems. Feel free to adapt it as needed.

However, there's lots of other functionality in iptables that we haven't talked about. Iptables is capable of limiting both the number of connections and the bit rate on those connections in both the inbound and the outbound direction. Iptables can do network address translation and even other types of packet mangling in-line. You can turn on the passive OS fingerprinting functionality and iptables will log its best guess of the type of machine when you tell it to LOG traffic. It's worth perusing the `iptables` manual page to get a sense of the different functionality that's available.

There are also lots and lots of iptables How-To guides on the internet. These include guides on how to turn a multi-homed Linux system into a network layer firewall, VPN gateway, etc.

Wrap-Up

Just a few parting words before we finish up today…

Looking Ahead

Coming Up Next:

- Rootkits and File Integrity Assessment
- Physical Attacks and Defenses
- Access Controls and Sudo
- Kernel Tweaking

Plus, plenty more fun on Day 3!

If everybody followed the guidance in today's course, most companies would be a lot more secure than they currently are. But there's so much more that we can talk about!

We covered some remote exploits today but didn't talk at all about what attackers do after they break into systems. So, tomorrow we'll look into exploit tools like rootkits a little and discuss tools that you can use to detect when attackers have dropped a rootkit onto your system.

We'll also talk a little bit about how somebody with physical access to a machine can break root on the box, as well as cover some defenses against common attacks. And as long as we're talking about physical access controls, we'll also talk about user-level access controls, password threats, etc. There's also some material on the sudo mechanism for granting limited root privilege to users who need it.

There are a number of system-level settings you can make to kernel parameters to harden your systems against certain kinds of spurious network traffic, local denial of service conditions, and even control core dump behavior. More on these settings tomorrow, too.

There's lots of fun to be had on Day 3 as well. Stay tuned…

That's it for Today!

- Any final questions?
- Please fill out your surveys!

Hal Pomeranz
hal@deer-run.com
http://www.deer-run.com/~hal/
@hal_pomeranz on Twitter

Lab Exercise

- Create host-based firewall policy
- Use automation to manage policy

Exercises are in HTML files in two places: On the course USB media, and in the home directory of the SANS user in the lab virtual machine. These two locations contain identical copies of the exercise files. It just depends on whether you prefer to look at them from a web browser inside the lab VM (because you're working in full screen mode) or from your host operating system.

1. Open your web browser
2. Select "File… Open…" from the browser menu (Ctrl-O is the usual keyboard shortcut)
3. This is Day 1, Exercise 3, so navigate to *.../Exercises/Day_1/index.html* and open this HTML file
4. Click on the hyperlink, which is the title of Exercise 3
5. Follow the instructions in the exercise

SEC506.2 | Securing Linux/Unix

SANS

Linux/Unix Hardening (Day 2)

Hal Pomeranz
hal@deer-run.com
http://www.deer-run.com/~hal/
@hal_pomeranz on Twitter

Today's Agenda

- Rootkits and File Integrity Assessment
- Physical Attacks and Defenses
- Password Attacks and Access Controls
- Sudo
- Warning Banners
- Kernel Settings for Security

Yesterday's course was all about defending against various kinds of remote exploits. Today, we'll focus in on more local sorts of exploitation and appropriate defenses.

When attackers gain access to your system, they'll likely plant a *rootkit* that allows them to maintain access and mask their activities on the system. The good news is that there are widely available tools that will allow you to detect these unauthorized system modifications, so we'll discuss those tools as well.

The ultimate "local attack" is an attacker with physical access to your system. There are some well-known ways to break root when you can physically touch the system, and we'll look at those and how to defend against them.

As long as we're talking about physical access controls, we may as well talk about access control in general. This includes well-known attacks against the Unix password system, managing root access in a more secure fashion using sudo, and even the appropriate content for those "warning banners" that always seem to pop up when you log into a system or network (and why these are so important).

Finally, we'll look at some other OS level settings—primarily kernel-based—that can be used to harden your system against certain kinds of malicious network activity or activity by the users on your system.

Post-Exploitation: Rootkits

Yesterday, we talked a lot about remote exploits. But what do attackers do once they're in? That's the subject of the upcoming section.

Back Doors

- Attacker wants to make it easy/quiet to get back in
- Often `sshd` replaced, but any networked binary will do
- Gives root shell for hard-coded password and/or IP source
- Mods sometimes detectable using `strings` command

One common scenario is deploying backdoors into already compromised systems. An attacker who breaks root on your machine probably doesn't want to go through whatever hassle was required to break root initially. Besides, the original root compromise may have been "noisy" enough that the attack was detected and the hole closed. However, if the administrator fails to detect the backdoor left behind by the attacker, then the attacker still has free access to the system.

The most common targets for backdoor type Trojans are the common remote login services like SSH, but bear in mind that any service listening on a network port could be backdoored. Sometimes, the Trojan horse gives a root shell if the attacker connects from a particular address and/or port; sometimes, there's a secret account and/or password encoded in the binary that gives root access. In the latter case, running `strings` on the binary sometimes reveals the offending username and password, though attackers are getting more astute about hiding such information in their binaries (including using tools to obfuscate their binaries and make reverse-engineering more difficult).

Attackers will also put "break root" backdoors in various set-UID binaries on the system (`su`, `passwd`, and even `ping`) in case they're limited to unprivileged access on the system.

rootkits

- Attackers also want to install DDoS tools, malware, et al
- Need much more intricate tools to cover their tracks
- rootkits are pre-packaged bundles of malicious software
- Rootkits may "harden" system and remove other rootkits!

Reasonably sophisticated attackers, however, want to be able to do a lot more than just log back into your system as root—they probably had a reason for compromising your machine in the first place. They may want to use your system as a proxy or a server for hosting their malware. They may want to use your resources to send spam or mine Bitcoin. They may install malware to steal credentials, payment card info, or anything else valuable moving through your systems.

An attacker usually wants to remain hidden on your system for as long as possible. To this end, they will compromise a number of operating system programs to hide their tracks. Bundles of compromised programs are packaged together into "rootkits" so that even inexperienced hackers can evade detection (for some sample rootkits, go to `packetstormsecurity.org` and search the archives for "rootkit"), but often contain easy-to-run installation scripts for kiddies. Generally, the rootkits contain software in binary form and are therefore OS dependent.

What's been interesting lately is that there seems to be a lot more competition between the various groups that are going around rooting boxes. The newer rootkits we're seeing actually contain some fairly sophisticated tools for finding and eradicating other rootkits that may already be installed on the system. The newer rootkits will even "harden" systems to close backdoors that somebody else might use to take over the box!

http://www.chkrootkit.org/links/ contains links to write-ups and discussions of classic Linux rootkits.

Hiding Files

- `ls`, `dir`, `find`, etc. all replaced to hide attackers' files
- Rootkits may use conf files to change behavior "on-the-fly"
- Binaries are crafted to have the same size (and same timestamps) as originals
- May use "immutable bit" to make files harder to remove

The typical rootkit will deliver Trojan versions of `ls`, `find`, `dir` and any other command that an administrator might use to detect the hidden files. In many cases, the Trojaned binaries will reference a configuration file that lists the files and directories that should be hidden. Sometimes, the malicious binaries will ignore files and directories that contain a particular string somewhere in the file or directory name.

Some recent rootkits are using executable compressors/packers to reduce the size of their binaries. That way, when the rootkit is installed on the system, the Trojaned binaries can be padded out to match the size of the original binaries. Of course, the timestamps on all installed files are also reset to match the original files. While this will fool novice admins, anybody using any sort of integrity checking tool (like AIDE or Tripwire) will easily be able to detect the switch.

In order to make things harder for admins, modern rootkits are setting the "immutable" attribute on installed binaries. While the superuser can remove the immutable bit with the `chattr -i` command, many novice admins don't know how to do this and will be unable to remove the infected files without reinstalling the OS. Note that you can use `lsattr` to detect files that have the immutable bit set.

Hiding Processes

- `ps`, `top`, `lsof` commands replaced with versions to hide attacker's processes
- Also need to modify `kill`, `pkill`, `killall` etc.

Rootkits also want to hide running processes (like the DDoS daemons and the rootkit's packet sniffer) from the administrator. Rootkits will contain a Trojaned version of `ps`, which hides certain processes—either specific processes or those listed in a configuration file. A Trojaned version of `lsof` may not only hide the attackers' files but might even contain a local root backdoor, since `lsof` is often installed set-UID or set-GID.

Similarly, the attacker doesn't want their processes to be stopped by the admin. The `kill`, `pkill`, and `killall` programs are "fixed" so that they wouldn't actually kill any of the attackers' processes.

Hiding Network Connections

- `netstat` and `lsof` will not show attacker connections
- Hide "promiscuous mode" on network interfaces
- Disable `syslogd` logging of attacker activity

Normally, when a machine is running a packet sniffer, the system administrator will be able to see that the network interface is in *promiscuous mode*. Most rootkits that contain a packet sniffer will attempt to hide this information from the administrator. In the old days, this meant replacing the `ifconfig` program (or the `ip` command under Linux), but you'll find that `ifconfig` no longer reports promiscuous mode on Linux, Solaris, and possibly other newer Unix-like operating systems.

`netstat` can be used to detect network connections in progress. Typical rootkits install a new `netstat` that not only hides the attackers' backdoor connections but also hides connections between the DDoS daemon running on the system and the master controller for the daemon.

Similarly, the rootkit will contain replacement Syslog agents that do not log the attacker's activity. Silent and deadly…

Problems Hiding Info with Trojans

Too many binaries to replace:

- Hard work
- Greater chance of detection
- Might miss one

Checksum tests (AIDE/Tripwire) allow for easy detection

As you might gather from the last few slides, there are a huge number of programs in a typical Unix operating system that need to be replaced by the attacker's rootkit. And the attacker has to keep their rootkit programs up-to-date with OS updates from the vendor, because if their programs start working differently from the OS-supplied versions, their rootkit will be detected by the users and administrators of the system. This is a lot of work.

Furthermore, programs like AIDE and Tripwire (discussed in the next section) can alert system administrators when a binary file has been modified by comparing a cryptographic checksum of the existing binary against a known good database of checksum values. Even built-in OS tools like the RPM utilities under Linux can be used for this (assuming the attacker hasn't replaced the `rpm` utility or the database it relies on).

In fact, past some point of diminishing returns, the more complete and complex a rootkit becomes, the *greater* the chance of detection by a careful system administrator. So, what's a poor attacker to do?

Kernel Level Rootkits

- Suppose you could replace the underlying kernel routines?
- Single modification corrupts every program on the system!
- Modern Unix systems support dynamic kernel modules– no reboot required!
- Some kernel rootkits can even attack via `/dev/mem`– no loadable modules!

For a while, there was quite a bit of active research in the area of kernel rootkits. The advantage to this approach is that by attacking the kernel in the appropriate way, you instantly and globally hide your exploits from the entire operating system at a very fundamental level. For example, suppose the attacker were able to compromise the kernel routine that reads directory entries—if the attacker were able to tell this routine to never display their hidden `/dev` directory, then no OS program could detect it, and an administrator would even have difficulty removing the directory with `rm`! Similar tactics could be used to interfere with the kernel routines that deal with file access, network sockets, etc.

Older Unix kernels could only be modified by being recompiled and the system rebooted under the new kernel image (watch for suspicious reboots). These days, Unix machines allow root to dynamically load and unload kernel modules—possibly replacing existing kernel routines with malicious code. The SUCKit rootkit (described in Phrack #58—see *`www.phrack.org`*) actually gets into the kernel via `/dev/kmem` and doesn't even require a loadable kernel module—so it can even work against kernels compiled without loadable module support! Yikes!

Interestingly, development of kernel-level rootkits seems to have largely died out. I'm not certain why this is because a truly well-written kernel rootkit would be an extremely powerful tool. They're even useful for the "good guys" because they allow you to create honeypot systems and hide your monitoring tools from the attackers via the kernel rootkit.

For a nice discussion of kernel rootkits, see *http://la-samhna.de/library/rootkits/index.html*. This document gives pointers to known kernel rootkits along with software that can be used to detect them, as well as discussing how such rootkits are implemented. The eNYeLKM rootkit for Linux 2.6.x kernels can be downloaded from *http://www.enye-sec.org/*. The Phalanx2 rootkit that is being used in many attacks can be found at *http://packetstormsecurity.org/search/?q=phalanx*

What Do You Do?

- Analyze system by first booting from clean OS
- Live analysis tools on read-only media where possible
- Don't try to "clean" systems—reinstall!
- Don't necessarily trust your backups

If you think your machine may have been compromised, the right thing to do is to take it out of production and bring it up in an isolated environment by booting off of your OS media (or simply remove the disk drive from the system and attach it to some other machine for analysis). This will guarantee that you have an uncompromised kernel image. Of course, if the attacker's intention was to hurt your business, they've just succeeded in at least taking your machine off the air for some period of time.

Similarly, bring over any analysis tools (AIDE et al) on read-only media if possible and always compare system binaries with binaries taken off of the OS media rather than some other system (which may have also been compromised).

When you get around to wanting to close the holes in your system, your best bet is to simply reinstall from scratch and then close the holes on a "virgin" system. Who knows, you might have missed a backdoor on the old OS image that would allow the attacker the re-establish access if you put the machine back into production without rebuilding. Reinstalling from backups may be tricky because you can't be certain exactly when your system was compromised.

rkhunter

Check for "signatures" of known rootkits

Looks for signs of:

- Binary modification (Trojans)
- Lastlog, wtmp, wtmpx deletions
- Sniffer logs and rootkit config files
- Loadable Kernel Module Trojans

Will use local OS binaries by default...

Note that rkhunter (*http://sourceforge.net/projects/rkhunter/*) can be run on a system that you believe may have been compromised, and will detect the signatures of many common rootkits (including some kernel-based rootkits). It can help locate hotspots of trouble and known rootkits on a wide variety of Unix-like operating systems. It does a fairly good job if all you are after is trying to determine that a system has been rooted, but it in no way replaces a thorough forensic investigation.

chkrootkit is another tool with similar functionality. Unfortunately, it appears that chkrootkit hasn't been updated since 2009 and therefore, is less useful. You can find the last version of chkrootkit at *http://chkrootkit.org/download/*

Summary – Trojan Horses

- Trojans heavily used for "backdoor" access and hiding
- Sophisticated rootkits widely available
- Integrity checking tools can be used to detect changes to filesystem objects
- Malicious kernels may be impossible to detect without taking system off-line

Rootkits have proliferated heavily as a "push-button" means of ensuring that an attacker can avoid detection and later regain access to the system as required. Pre-compiled rootkits for a variety of popular OS flavors are easy to obtain, install, and customize.

However, specialized integrity checking tools like AIDE, Tripwire, or even built-in OS functionality like the Linux RPM utilities can be used to detect when system binaries have been modified. If used properly, these tools will alert the administrator to rootkit installs and make corrective action easier (because the admin knows *exactly* which files have changed).

In the internet security arms race, though, the growing use of malicious kernel modules gives the attacker the ability to subvert the system kernel and globally change the behavior of all applications running on the system (including the integrity checking tools that the administrator relies on for alerts). Without taking the system off-line and booting off a "known good" kernel, it may be impossible to detect a sophisticated kernel exploit of this type.

Mitigation: AIDE

AIDE (*Advanced Intrusion Detection Environment*) is a filesystem integrity checking tool—it can alert you when critical files on your system have been changed, such as executables, libraries, and configuration files in `/etc`. When system files are changed without warning, it could be a sign that an attacker has compromised your system and installed a rootkit or other malicious software.

Note, there's another popular Open Source file integrity checking tool out there called Samhain (*http://la-samhna.de/samhain/*). The commercial product in this space that everybody is familiar with is Tripwire (*http://www.tripwire.com/*).

How it Works – Overview

- Create config file listing critical files/directories to watch
- Generate initial file/checksum database for this list of files
- Periodically re-run AIDE:
  - Compare current file/directory info to database
  - Report discrepancies

AIDE works by initially creating a database of information about files on your system along with checksums it computes on the contents of these files. Files are added to this database based on a configuration file created by the administrator. The database for a machine should be created as soon as the machine is installed and *before* it gets connected to a production network where it might be compromised.

The administrator then runs AIDE on a regular basis (nightly out of `cron` is a fairly common implementation). AIDE compares the current state of the file with the parameters and checksums stored in the initial database. If there are any differences, the differences are reported.

Obviously, sometimes changes to files are normal—`/etc/shadow` changes when a user updates their password, binaries change when you install a patch from your OS vendor, device files are added when you add more hardware to your system, etc. AIDE will continue reporting that files have changed until you update your original database to reflect the new state of the file. Thus, it's a good idea to only have AIDE watch files that don't tend to change that often—luckily, this describes most of the critical files in your operating system. Most of the work deploying AIDE involves creating and tuning your configuration files to eliminate false positives.

What Problem Does it Solve?

- Lets you know *exactly* which files changed on your system
- This is indispensable information after a security incident
- However, the greatest recurring value may be alerting you to mistakes by local admins

When somebody has broken into your system, you're often left wondering (a) what files did they change and what backdoors did they leave behind, and (b) when did they change these files (aka when was my last good backup)? If run correctly from the moment the system is installed for the first time, AIDE can answer both of these questions with complete assurance.

While it's likely that some of your systems will get broken into at some point, this is hopefully a fairly rare occurrence. The other 364 days out of the year, AIDE is really good at letting you know when your local administrative staff makes mistakes—either installing patches and clobbering files that they shouldn't or making configuration file updates that they shouldn't. AIDE's greatest long-term value to an organization may be its usefulness as a change control and configuration management tool.

The Problem

Attacker who roots your box can modify your AIDE install

Solutions include:

- Binary and database on local, read-only media
- Read-only share from central, protected host
- Remote checks via SSH from central host
- Read-write local access with periodic external verification

The one issue to consider when deploying any integrity-checking tool is the security of the database and binaries used by the local system during the integrity audit. When an attacker gets in and roots your machine, they could potentially install a rootkit and then modify your integrity database to reflect the new state of the compromised files. Or, an attacker might just install a compromised version of the AIDE software that never reports any problems with the files that they install.

One solution is to burn your database and binaries onto a piece of read-only media like a CD-ROM (assuming you have a read-only device available to the system). The only problem is that when you update the database, you need to burn a new CD—which can be time-consuming on a big network, but maybe not so bad for a small group of very critical machines.

Some sites keep their databases and binaries on a central file server and then export that filesystem read-only. This solution can work pretty well on protected internal networks, though theoretically, an attacker could manipulate the data stream from the file server to get the system to run their bogus version of AIDE or read their bogus database. Of course, you're completely hosed if the attacker compromises your file server.

Recently, I've been experimenting with running my integrity checks from a central security server via SSH. The security server goes to each machine in my network, and in turn, copies over the AIDE binary and database, runs the scan and then pulls everything off the remote system. Unless the attacker happens to be on the system when the scan is running, there is no sign that the system is even being monitored! We'll look at this solution in more detail tomorrow.

Some sites just say, "the heck with it" and keep read-write copies of their binaries and database on the local machine. If the attacker is not aware that AIDE is being run or does not know how to circumvent AIDE, then life is good. Still, if you're going to be running in this mode, it's a good idea to keep a copy of your database

and binaries on some other machine and run periodic comparisons to see if either has been modified on the local system. In fact, it's generally a good idea to run complete audits on a system at random intervals to protect against attackers who "hide" their backdoors during the regular nightly cron job that runs AIDE on the box.

Note that the commercial tool Tripwire gets around these problems by digitally signing the Tripwire agent and database from a master administration console. The digital signatures must verify correctly in order for the agent to function. There's no particular reason why you couldn't engineer a similar solution on your own using AIDE with PGP.

Installation Notes

Includes standard "`configure`" script

Large number of Open Source dependencies:

- GNU `bison`, `flex`, and `make`
- Zlib data compression library
- mhash library (checksum algorithms)

Source tweaks may be required for non-mainstream OSes

The AIDE distribution comes with a standard "`configure`" script, so normally you'd think that the build would be pretty straightforward. However, the AIDE developers clearly work on Linux systems where the standard build tools are GNU tools like `bison`, `flex`, and GNU `makes`. On commercial Unix operating systems (Solaris, HP-UX, etc.) these are typically add-on packages that must be installed first. AIDE also requires the zlib compression library.

The mhash library supplies additional file checksum algorithms, but is not strictly required. If mhash is not present at compile time, AIDE will still be able to compute md5, SHA-{1,256,512}, rmd160, and tiger checksums, which is usually sufficient. mhash adds haval, gost, whirlpool, and CRC32.

Here are some URLs where all of these tools can be obtained:

GNU Software: `http://www.gnu.org/`

Zlib: `ftp://ftp.info-zip.org/pub/infozip/zlib/`

Mhash: `http://mhash.sourceforge.net/`

Also note that while AIDE seems to build fine on "mainstream" platforms like Linux and Solaris, I've had to make tweaks when building on other Unix platforms. Cross-platform support is gradually improving, though.

Note that there really isn't a "`make install`" target for AIDE. The build creates a statically linked "`aide`" binary in the "`src`" subdirectory—just copy this binary anywhere.

aide.conf – Per File Checks

p	Permissions/mode bits
i	Inode number
n	Number of links
u	File owner (user)
g	Group owner
s/b	File size in bytes/blocks
S	Checks that file is growing
a/m/c	Access/modify/inode timestamps

In the AIDE config file, `aide.conf`, you must list the files you want to watch and the parameters you want to watch for each file. There are several basic file parameters that AIDE can report on, and we'll see how to use these flags in a minute.

- "`p`" means report on the current setting of the file mode and complain if somebody has `chmod`-ed the file.
- "`i`" means report if the inode number has changed, which indicates that the file has been replaced.
- "`n`" means report on the link count of the file—this is particularly useful on directories because the link count goes up when subdirectories are created in a given directory (due to the "`..`" link in the subdirectory, which points back to the parent).
- "`u`" and "`g`" report if the owner or group owner of the file is changed, respectively.
- "`s`" watches the file size in bytes, while "`b`" looks at the number of data blocks used by the file. These values tend to change when a new version of a program is installed.
- The "`S`" parameter is used to keep track of files (typically log files) that are expected to get bigger and bigger, but never shrink. Shrinkage may indicate that somebody has deliberately edited the file to remove information.
- "`a`", "`m`", and "`c`" report on the last access, last modified, and inode change timestamps, respectively. "`a`" is used rarely because most files are being accessed all the time—even "out of the way" files will get read by your nightly backup routines. Hence, monitoring the last access time does nothing but produce a ton of false positives.

Depending on the system where you're using AIDE and the compile-time options, you may also be able to use "`acl`" to monitor (POSIX) ACLs, "`xattr`" to watch extended attributes, and even "`selinux`" to monitor SELinux contexts.

aide.conf – Checksums

- Basic checksums include `md5`, `sha1`, `sha256`, `sha512`, `tiger`, and `rmd160`
- Multiple checksums on "critical" files for maximum security
- Single checksum on normal files to reduce system impact

The administrator must also specify which checksum(s) should be run on each file. AIDE can actually use any of several different checksum algorithms represented by the keywords shown above.

The most secure configuration is to run at least two checksums on a given file. It may be possible (at least for some checksumming algorithms) to create a Trojan horse binary that has exactly the same checksum as the original binary. However, even if an attacker were able to generate a binary that could fool any single checksum algorithm (and this has not even been demonstrated for most of the above algorithms), it seems *extremely* unlikely that they could create a single binary that would fool *two different* algorithms.

As we will see, AIDE's default is to run only the MD5 algorithm against files. This is so that normal integrity checks (which might look at 10-20K files on a given system) don't bog down the machine too much. However, it is prudent to identify "critical" files on the system and monitor such files with multiple checksum algorithms as a special case. More on what constitutes a "critical" file shortly.

aide.conf – File Entries

Specify file regexp and list of parameters:

```
/usr/bin/su$   p+i+n+u+g+s+m+c+md5
```

Common sets have pre-defined macros:

R	p+i+n+u+g+s+m+c+md5 ("read-only")
L	p+i+n+u+g ("log file")
>	p+i+n+u+g+S ("growing log file")
E	Empty set ("ignore *everything*")

The format of most of the lines in `aide.conf` is an `egrep`-style regular expression that matches one or more files and directories followed by a list of parameters and checksums that should be monitored for the given file(s). Note that the regular expressions in the first column are implicitly rooted at the beginning of the file pathname—in other words, there's an implied "^" at the beginning of each regexp and there's nothing you can do to change that. However, when referring to a specific file or directory, it's usually very important to end the regexp with a "$" to indicate the end of the pathname so that you don't end up matching too much.

Since it's a pain to keep typing a dozen letters and digits for each file entry, AIDE defines some macros for common sets of configuration options (and you can even define your own, as we'll see later). 98% of the entries in your configuration will use "R", which is the default "check everything but last access time and use MD5 checksums" configuration that's appropriate for most files on the system that don't get updated except by patches (binaries, scripts, and even most configuration files). You can use "L" on (some) log files and other files that change regularly (like `/etc/passwd` and `/etc/shadow`). This macro checks everything but the file size and last access and last modified times, and does not run any checksums to attempt to verify the file contents. However, it will report if the file is recreated or if the permissions or owner/group owner change. There's also a special ">" class that's supposed to be used on files that should change but never shrink.

"E" is the empty class (*ignore everything*) that can be used for exceptions—"watch all files in `/etc` except for `/etc/syslogd.pid`, which I don't care about." Actually, AIDE gives you a different way of handling this that's probably better—more on this on the next slide…

aide.conf – Directories

- By default, AIDE recursively descends through directory trees, catching all entries
- Use !/= to modify this behavior:

```
=/usr$                    R   # check /usr itself,
                              # but don't recurse

/etc/namedb               R   # watch zone files
!/etc/namedb/slave            # but not slave files
```


Since the filename entries in `aide.conf` are regular expressions, specifying a directory name with no trailing "`$`" at the end of the pattern will match everything beneath that directory as well, meaning AIDE will monitor all files and subdirectories inside the directory you specify. Even if you terminate the regular expression with a "`$`", AIDE's default behavior is to recursively descend into directories and monitor all files. In many cases, this is what you want, but not always.

Putting an = sign before a directory name in `aide.conf` means to check the directory entry itself, but don't try and recurse into that directory. In our first example, we want to watch the `/usr` directory file (to make sure that no new subdirectories are being added), but we don't want to check all of the files and directories under `/usr` (though we'll probably have specific configurations for some directories like `/usr/bin` and `/usr/local/bin` elsewhere in the `aide.conf` file). Note that it is *vital* to terminate this regexp with a "`$`", or else you'll still end up matching everything under `/usr` and putting all of those files into your database!

You can also use `!` to exclude files and directories (and all subdirectories below a given directory) from consideration. It's always a good idea to watch for updates to your DNS zone files, but you don't want to get a report from AIDE every time your machine downloads a slave zone from the master DNS server. Note that the "`!`" syntax seems to be a better way of dealing with exceptions than the "`E`" macro we discussed on the previous slide.

Partial `aide.conf` File

```
database=file:/var/aide/aide.db.gz            # where DB lives
database_out=file:/var/aide/aide.db.new.gz    # put new DB here
gzip_dbout=yes                                # support GZIP DBs
verbose=20                                    # 0-255
H = p+i+n+u+g+s+b+m+c+sha512+rmd160           # "heavy" auditing

/dev                                 L        # watch /dev entries
!/dev/[pt]typ[0-9a-f]$                        # these change a lot

/root                                H        # critical area
/root/.ssh/known_hosts$              >        # this file changes

=/etc$                               L        # critical directory
/etc/.*                              H        # watch contents
!/etc/ntp.drift$                              # ignore this file
```


Here's a sample of the configuration rules and some other functionality that can be included in `aide.conf`. The first line tells AIDE where the integrity database is located on the system (the AIDE developers are working on being able to store integrity information in other formats besides plain text files, hence the "`file:...`" syntax). The second line is where AIDE should create new and updated databases when requested. The administrator must manually overwrite the old database with the new one—this is considered good practice. Newer versions of AIDE support GZIP-ed database files, but `gzip_dbout=yes` must be set.

"`verbose`" controls how much output you get. Unfortunately, the verbosity levels are not well defined at this point—0, 5, 20, and 255 seem to be the significant verbosity settings, with 20 being a good default.

The fifth line shows how you can define your own macros (and you're not just limited to a single character; you can use longer names if you want). Here, we're defining a "heavy" auditing macro for critical files that check the usual stuff that "`R`" does, but also adds the "`b`" (block count) setting and uses two stronger checksum algorithms rather than MD5.

You definitely want to keep an eye on your `/dev` directory because attackers love to hide files here. However, the ownerships of the `pty` devices change every time somebody logs in and logs out, so watching these generate lots of false-positives.

Similarly, the files in root's home directory are attractive to attackers as a possible backdoor, so we want to watch these. However, the SSH `known_hosts` file changes every time root logs into a new system, so we have to be careful how we monitor this file.

`/etc` is clearly one of the most critical directories to watch on the system. However, there are files like `/etc/ntp.drift` that are being recreated all the time under `/etc`. So, while we can assume that most files under `/etc` shouldn't change, we have to be careful how we monitor the directory itself and certain files in it.

Files/Directories to Watch

- "Significant" directories like /, /usr, /var, /dev, /tmp...
- Config files in root's home directory
- /etc (but beware of derived files in /etc)
- Crontab files and directories
- Kernel and boot loader (if any)

There are some sample `aide.conf` files in the doc directory of the AIDE source distribution. Other sample files for various operating systems can be found at:

```
http://www.deer-run.com/~hal/aide/
```

Feel free to use these as a starting point for your own configurations.

Certain files and directories should *always* be checked. These include:

- Directories like /, /usr, and /var shouldn't change under normal operation (nobody should be adding new files and subdirectories immediately under any of these top-level directories). Other directories like /dev (and /devices on SYSV-type systems), /tmp, and /var/tmp may change, but you should still use "L" to monitor the ownerships, permissions, and link counts (which will tell you if new directories are being created).
- Keep an eye on root's dot-files—.rhosts, .shosts, .profile, etc. The .ssh directory for root is very important to watch (especially the authorized_keys file), but be careful of the known_hosts file mentioned earlier and the prng_seed and random_seed files that get regenerated regularly.
- Always watch configuration files in /etc. You need to avoid files like syslogd.pid and named.pid, though, which change all the time. Ditto for files like mnttab and sharetab, which are generated from other configuration files at boot time.
- Keep an eye on your crontabs—you don't want anybody adding jobs without your say-so!
- Your kernel executable file and/or directory (typically something like /unix or /kernel or /bsd, etc.) including the directory that holds your loadable kernel modules (if your kernel supports dynamic linking). On BSD systems, you should also keep an eye on /boot, which is the initial kernel boot loader.

Also Watch `bin` & `lib` Dirs

- Monitor *all* `bin` and `lib` dirs on the system (including third-party software dirs like `/opt` and `/usr/local`)
- Modern systems can handle multiple checksums, even when scanning large numbers of files

It's vital that you watch *all* `bin` and `lib` directories on your system. On most Unix machines, this is a surprisingly large collection of directories including not only `/usr/bin` and `/usr/lib`, but also `/sbin` and `/usr/sbin` and `/usr/libexec`, OS-specific directories like `/usr/ccs` and `/usr/ucb` on Solaris, directories containing X windows tools like `/usr/X11` or `/usr/openwin` or `/usr/dt`, and third-party software directories like `/usr/local` and `/opt`. To find all the `bin` and `lib` directories on your particular flavor of Unix, run the following command (and prepare to be surprised):

```
find / -type d
   \( -name bin -o -name sbin -o -name lib \) -print
```

There can be tens of thousands of files in these directories covering hundreds of megabytes of disk space.

AIDE's "R" macro only computes a single checksum because computing multiple checksums used to be too resource intensive on older systems. Modern systems can normally handle computing multiple strong checksums on all files being scanned. I would go with this as a default unless the scan causes a significant performance impact on the system.

Don't Forget "Content" Dirs!

- Web server doc trees and CGI bins
- Anonymous FTP areas, file shares
- DNS zone files

People who've had their websites defaced know it's extremely useful to have your integrity database contain checksums for all of your web content and CGI scripts. Similarly, it's nice to run AIDE against your anonymous FTP archives to make sure you're not distributing Trojaned software without your knowledge and that your upload directories are not being used for warez.

Make sure your `aide.conf` file includes the directories where you keep your DNS zone files and any other shared network databases (LDAP, etc.).

The Problem with Log Files

- Monitoring log files seems like an obviously good idea
- Problem is that log files get moved, "rotated", and archived
- Watching logs just generates "noise"

You might think you would also want AIDE to watch all of your log files, so you can know if an attacker deletes your logs to cover their tracks. Unfortunately, the log rotation script that you run on your system also deletes and recreates new log files all the time, so you tend to get lots of false positives. There's not a lot of useful information you can get from monitoring your log directories, so you're better off just leaving them out of your configuration files.

Using AIDE

Generating your database:

```
# aide --config=/var/aide/aide.conf --init
  [… some informational messages not shown …]
# mv /var/aide/aide.db.new.gz /var/aide/aide.db.gz
```

Running a check:

```
# aide --config=/var/aide/aide.conf --check
AIDE, version 0.14
### All files match AIDE database. Looks okay!
```


Once you've created your initial configuration file, run `aide --init` to generate the integrity database. The location of your config file can be specified on the command line with the `--config` option (the default is `/usr/local/etc/aide.conf`). The new database will be created using the pathname you specified with the `database_out` parameter in `aide.conf`—make sure to move this directory to the location you specified with the `database` parameter. It's also a good idea to make a copy of the database in some other secure location so that you can spot comparisons between the archived copy and the running copy on the system. If you're going to burn your database onto read-only media, now's the time to do it.

Once you've generated the database and put it in the correct location, use the command "`aide --check`" to run a scan. Actually, the `--check` option is not required since AIDE's default is to run a scan using the specified configuration file. Notice that AIDE generates output even when there's nothing wrong—this is actually something of a problem when you're running AIDE from `cron`, so more on this later.

The next page shows a complete report from AIDE when some files on the system have changed. There's a summary at the top and a list of the changed files, and then a detail section of the report showing the parameters that are different for each file. Notice, for example, that the contents of `/etc/security/audit_data` have been changed (`mtime` and `ctime` are updated and the checksums no longer match), but the `/etc/dhcpd.leases` file has been completely recreated (new inode number and totally different file size).

```
# /var/aide/aide --config=/var/aide/aide.conf
AIDE found differences between database and filesystem!!
Start timestamp: 2006-03-11 11:49:56

Summary:
  Total number of files:        10822
  Added files:                  0
  Removed files:                0
  Changed files:                5
---------------------------------------------------
Changed files:
---------------------------------------------------
changed:/etc/cron.d/FIFO
changed:/etc/mail/statistics
changed:/etc/security/audit_data
changed:/etc/dhcpd.leases
changed:/etc/dhcpd.leases~
---------------------------------------------------
Detailed information about changes:
---------------------------------------------------
File: /etc/cron.d/FIFO
  Mtime     : 2004-03-21 12:14:02             , 2004-03-21 14:14:35
  Ctime     : 2004-03-21 12:14:02             , 2004-03-21 14:14:35

File: /etc/mail/statistics
  Mtime     : 2004-03-21 13:47:55             , 2004-03-21 16:02:57
  Ctime     : 2004-03-21 13:47:55             , 2004-03-21 16:02:57
  MD5       : Vhbdo2DxMxuwRZJE9+61OA==        , rc5K7XRiUfKJ0cET3jATYg==
  SHA1      : 8JkRx12+8u6/RrxevzPraGQZIdU=    , O20pi+SSSmAej/PraA/vwgJau+4=

File: /etc/security/audit_data
  Mtime     : 2004-03-21 13:00:00             , 2004-03-21 16:00:00
  Ctime     : 2004-03-21 13:00:00             , 2004-03-21 16:00:00
  MD5       : etieNZuZ7a7tP2oyeSQIEw==        , Jcxv+NP2V0h6KzXUuz55Dw==
  SHA1      : JvIDZbhcHfSoHYK0kKNKCDnPgRI=    , zM8M8kfjVZSK42xwpBEQXoAOLsA=

File: /etc/dhcpd.leases
  Size      : 4830                            , 4829
  Mtime     : 2004-03-21 09:44:46             , 2004-03-21 14:17:02
  Ctime     : 2004-03-21 09:44:46             , 2004-03-21 14:17:02
  Inode     : 66092                           , 66191
  MD5       : hHNVt5l4jTqbJ6EBdAoktQ==        , hLbyNZCAwmNp9pZuxMAtFw==
  SHA1      : vubdr2ZfdL3twvNMCD23+y77PYc=    , bm3hrIQBE9wxcA3kUp6o7eYLQ34=

File: /etc/dhcpd.leases~
  Size      : 5252                            , 5064
  Bcount    : 12                              , 10
  Mtime     : 2004-03-21 09:44:46             , 2004-03-21 14:17:02
  Ctime     : 2004-03-21 09:44:46             , 2004-03-21 14:17:02
  Inode     : 66091                           , 66092
  MD5       : 1XTlF0mmqOP9OjWnh/3Jkw==        , 1tUwsKKpJIJfvhQk6pz62w==
  SHA1      : kHJof80VuszZqIhrFV5ElUAXNOo=    , 38XGk7fI8SPwiAGBifg+WKhZFnI=
```

Thoughts on Automation

- You want to run AIDE from `cron`
- You don't want to get spammed if everything is OK
- Simple script hides normal output, shows real warnings
- Run periodic manual audits to detect tampering

Once you've tuned your configuration file, the usual protocol is to arrange for AIDE to run nightly from `cron`. The problem is that AIDE creates output whether or not there are any changes on the system—in most cases, you just want AIDE to shut the heck up unless there are problems.

Probably the easiest thing to do would be to create a simple little wrapper script like the following:

```
#!/bin/sh
TEMPFILE=/var/aide/.out$$

/usr/local/bin/aide --config=/var/aide/aide.conf --check \
         > $TEMPFILE 2>&1
if [ ! "`grep '### All files match' $TEMPFILE`" ]; then
     cat $TEMPFILE
fi
rm $TEMPFILE
```

Of course, you might get no report from `aide` for other reasons, like `cron` isn't working or an attacker has turned off `aide` on your system. You had better check periodically to make sure things are still working. In fact, it's a good idea to run manual audits at random times. Also, if you keep a read-write copy of your database on the local system, make sure you periodically run a `diff` between this database and your archived copy. Ditto for the `aide` binary itself.

Updating Databases

- Files will change during the lifetime of a system and database must be updated
- Use "`aide --update`" to run a scan and simultaneously produce new database
- Carefully check report before overwriting old database!

It's natural for files to change during the lifetime of a system, and you need to be able to update the entries for those files to get AIDE to stop whining at you. AIDE has `--update` mode that generates a report just like the `--check` option, but also dumps out a new database to whatever location you've specified using the `database_out` parameter in `aide.conf`. That way, after you've carefully checked the report to be sure that the changes are "expected", you can just copy the new database over top of the old one.

In fact, I will always run AIDE in `--update` mode, rather than using `--check`. That way, after I've reviewed the report, I have a new database waiting there that reflects the changed state of the system. Of course, you have to somehow ensure that the database written by `--update` was not tampered with while it was being written or at some point after that.

Summary Info – AIDE

Homepage, docs, download
http://aide.sourceforge.net/

Sample configs and SSH scripts
http://www.deer-run.com/~hal/aide/

AIDE was originally developed in Finland by Rami Lehti and Pablo Virolainen. It's been through several maintainers, but is now being hosted at SourceForge.

My tools and sample config files are available at my Deer Run website. We'll discuss my scripts for running AIDE via SSH tomorrow.

Lab Exercise

- Getting to know AIDE
- Goal is to understand how to configure and use AIDE, dealing with false-positives

Exercises are in HTML files in two places: On the course USB media and in the home directory of the SANS user in the lab virtual machine. These two locations contain identical copies of the exercise files—it just depends on whether you prefer to look at them from a web browser inside the lab VM (because you're working in full-screen mode) or from your host operating system.

1. Open your web browser
2. Select "File… Open…" from the browser menu (Ctrl-O is the usual keyboard shortcut)
3. This is Day 2, Exercise 1, so navigate to *.../Exercises/Day_2/index.html* and open this HTML file
4. Click on the hyperlink, which is the title of Exercise 1
5. Follow the instructions in the exercise

Physical Access Attacks

In this section, we'll look at simple "break root" type attacks when the attacker has physical access to a Unix machine.

Physical Access = `root`

- Single-user boot
- Boot off of OS Media
- Corrupt root filesystem
- Steal the disk drives!

It's pretty well understood that an attacker with physical access can find a way to get root privileges on a machine. Rebooting into single-user mode, booting off of OS media, and even the old "corrupt the root filesystem" trick all require one or more reboots of the machine, however. So, it's a good idea to keep an eye on your system logs for suspicious reboots.

As we'll discuss in the next few slides, there are some steps you can take to prevent many of these attacks. However, even if there were a way to make the machine completely safe from an attacker with physical access, the attacker could still forcibly open the machine and remove the disk drives. The drives could be taken away and mounted on some other system, though using an encrypting filesystem could help here. For similar reasons, also protect your backup tapes, because stealing a backup tape is likely to be a lot less tamper-evident than stealing the machine's disk drives!

By the way, if you're interested in encrypting filesystems, the standard option for Linux is the dm-crypt code (in the Linux kernel since 2.6) with the LUKS `cryptsetup` front-end (*http://en.wikipedia.org/wiki/Dm-crypt*). After the Truecrypt project was abandoned, two new projects have risen to take its place:

CipherShed: *https://ciphershed.org/*
VeraCrypt: *https://www.veracrypt.fr/en/Home.html*

Single-user Boot

- Single-user administrative shell has full root privileges
- Require root password before single-user shell is standard
- Interestingly, pre-**`systemd`** Red Hat OSes did not

Single-user mode was originally designed as a low-level administrative mode for taking backups, applying patches, and performing other tasks where the system should be "at rest" with no users and a limited number of processes running to disturb the state of the system. The single-user shell is a full root shell, so getting single-user access to the system would allow an attacker to plant backdoors on the system at will.

For this reason, typical Unix systems require the user at the console to enter the root password before getting access to the single-user shell. Surprisingly, Red Hat-based operating systems historically did not require a password before entering single-user mode (the only modern operating systems your author is aware of that did not do this). After talking with some people who've gone through RHCE training, the "official" position from Red Hat seems to be that you're supposed to block access to single-user mode by using a bootloader password, which we'll talk about in the next few slides. Frankly, I think this is a little wrong-headed, but let's talk about bootloader passwords a bit and then you can make up your own mind.

The cutover to `systemd` in RHEL7 meant that Red Hat was forced to join the modern world—prompting for the root password when entering single-user mode is now the default. See the `/usr/lib/systemd/system/rescue.service` unit file for the startup action that prompts for the root password. In RHEL6 you can force the root password at single-user boot by setting `"SINGLE=/sbin/sulogin"` in `/etc/sysconfig/init`. For RHEL5 and earlier, just add `"sum:S:wait:/sbin/sulogin"` in `/etc/inittab`.

Boot Off OS Media

1. Boot off OS media or rescue disk
2. Exit install program to shell prompt
3. Mount local drives
4. Insert favorite back door
5. Reboot normally

Use bootloader password to control system boot options...

When administrators have forgotten the root password for a machine, the typical way to get access is by booting off of the original OS media. Of course, an attacker can also use this approach.

Booting off of OS media generally puts you into the system-install program, but there's usually an option for dropping out to a shell prompt. This is effectively a root shell, albeit in the limited environment used by the install program. However, once at this shell, the user on the console can mount the local disk drives from the machine and have full root access.

For administrators, the easiest thing to do at this point is to blank the password field for the root account in `/etc/shadow` and reboot normally. However, any number of backdoors are possible, such as setting the set-UID bit on one of the shells, or on an editor that allows shell escapes, etc. Once the system is rebooted off of the normal system image, the backdoor that was created can be used to access the machine with root privileges.

Setting a bootloader password means that the system will require the user on the console to enter that password before any "special" boot commands can be used—like booting from CD-ROM, or even booting into single-user mode. The trick is to still allow the system to boot using the normal boot command so that it can reboot normally in the event of a power outage or other temporary failure. More on this in just a moment.

Bootloader Security

Can set up boot-level passwords:

- Stops booting the system and other media unless password entered
- Need to allow the system to reboot automatically after a crash

Attacker with physical access can still steal your disk drives…

You can block booting off of CD-ROM (and booting into single-user mode) by setting password protections on the boot loader. The trick here is to make sure you do this in such a way that the system can reboot automatically after a crash—you don't want to have to come in at 3 a.m. to type in the boot password on your mail server when the kernel panics. Most boot loaders have a mode that allows normal reboots off the default boot device, but require a password whenever somebody on the console enters a non-standard boot command (like booting from CD-ROM or into single-user mode).

Even if you get this right, though, somebody with physical access to your system can just rip the disk drives out of your machine and mount them up on some other system that doesn't have these sorts of protections. On the other hand, you'd probably notice if this happened to one of your machines, so at least you'd know you had a problem.

There's also a problem with bootloader passwords and typical Intel-based systems…

Intel Architecture Problems

- On Intel, system BIOS generally invokes boot loader
- Attacker can force BIOS to look at alternate boot device
- This does an "end-run" around any bootloader security
- BIOS config password prevents tampering with boot

The problem with bootloader security on Intel-based machines is that the system BIOS generally triggers the boot loader. However, the BIOS can be interrupted and told to boot off of an alternate device, such as the CD-ROM drive or a floppy. This means the boot password you set on your hard-drive-based boot loader will be completely ignored. And once the attacker successfully boots from CD-ROM, they can just mount the filesystems from the local disk drive.

It is possible to set a BIOS password that must be entered as soon as the system powers on. The problem is that a BIOS password generally prevents the system from rebooting automatically. BIOS passwords can be helpful on laptops and other personal machines but are generally not advisable on server-type machines.

Modern BIOSes have two levels of passwords—one protects the BIOS settings from tampering, and the other is a "power-on" password that must be entered before the system can boot. In our case, simply protecting the BIOS settings would be sufficient, so use the BIOS "configuration password" feature if your BIOS supports it.

Boot Passwords

Don't use the root password!

- Boot password could be stolen (easily)
- Don't forget to update boot password when root password change occurs

Primary goal of boot-level passwords is to thwart "outsiders"

Just pick a "well-known" password and stick with it

The question with these bootloader passwords is what password to use. Some sites use the system root password as the bootloader password, but this is actually a *bad idea*. One problem is that bootloader passwords are often stored insecurely on the system (sometimes in cleartext form in configuration files). The other problem is that when a site updates their root passwords, they tend to forget to update the bootloader password. Six months later, you're sitting at the system console trying to remember an old root password so you can boot the system off CD-ROM for an upgrade.

These bootloader passwords are primarily targeted at preventing outsiders from breaking into your system due to physical access. That being the case, it's OK to use the same bootloader password across all of your systems and give it out to all of the administration staff who need to know it. This seems like the best policy.

The GRUB2 Bootloader

- Create password hash with `grub2-mkpasswd-pbkdf2`
- In `/boot/grub2/grub2.cfg`:

```
set superusers="root"
password_pbkdf2 root grub.pbkdf2.sha512.10000.BA12C…
```

- Enter user/password to access GRUB commands at boot
- Set secure permissions on `grub2.cfg`!

To password-protect your GRUB installation, you will need to update your `/boot/grub2/grub2.cfg` file. The important configuration lines look like this:

```
set superusers="root"
password_pbkdf2 root grub.pbkdf2.sha512.10000.BA12C19A275EEB3FAFBA...
```

This creates a user called "`root`" with a hashed password entry. You are not required to use the name "`root`." This user will be used to access commands at boot time only and can use any name you want. This user has nothing to do with normal user logins on the system.

You can generate the password hash with the `grub2-mkpasswd-pbkdf2` command:

```
# grub2-mkpasswd-pbkdf2
Enter password: (not echoed)
Re-enter password: (not echoed)
PBKDF2 hash of your password is grub.pbkdf2.sha512.10000.BA12C19A275EEB3FA…
```

Once you've set the GRUB2 password, the behavior of the GRUB2 splash screen that appears when the system boots is changed. While you can still automatically boot the default boot choice, all other GRUB2 commands (like "`e`" to edit the boot command) will trigger a username/password prompt.

Even though the password is now encrypted with a one-way hash function, it's still a good idea to make sure the `grub2.cfg` file is only readable by the superuser, just to make it harder for an unprivileged user to steal the password hash and crack it offline.

Corrupt Root Filesystem

- Repeatedly power-cycle system
- Root filesystem will eventually become inconsistent–manual `fsck` required
- System provides root shell without asking for a password
- Attacker can `fsck` filesystem, change root password, etc.

Booting the system off of OS media or a rescue disk requires the attacker to come prepared with a bootable disk, and booting to single-user mode usually requires the root password. The method presented here requires no special preparations or passwords.

Simply walk up to a Unix machine and crash it—flick the power switch, pull out the power cord, etc. Now turn the machine back on and let it boot normally. If it gets past the filesystem checks successfully, crash the machine again. Eventually, the root filesystem of the machine will become corrupted and the user on the console will be prompted to manually `fsck` the filesystems. The system then spawns a root shell. Since the root filesystem is corrupt, the machine can't necessarily read the `/etc/shadow` file, so no root password is required. This isn't single-user mode! Once the filesystems have been `fsck`-ed, the attacker can now blank the root password or install some other backdoor before bringing the machine up in multi-user mode.

It's interesting to note that at least Red Hat and Solaris releases actually do prompt for the root password before giving the root shell for the manual `fsck` (even though, as noted earlier, RedHat doesn't prompt before giving the single-user shell—go figure). While this means that the user could potentially be locked out if, in fact, the `/etc/shadow` file did get corrupted, perhaps this "fail closed" stance is better than the alternative.

Lab Exercise

- Boot loaders and single-user mode
- Practice your GRUB configuration skills
- Get booted into single-user mode

Exercises are in HTML files in two places: On the course USB media and in the home directory of the SANS user in the lab virtual machine. These two locations contain identical copies of the exercise files—it just depends on whether you prefer to look at them from a web browser inside the lab VM (because you're working in full-screen mode) or from your host operating system.

1. Open your web browser
2. Select "File… Open…" from the browser menu (Ctrl-O is the usual keyboard shortcut)
3. This is Day 2, Exercise 2, so navigate to *.../Exercises/Day_2/index.html* and open this HTML file
4. Click on the hyperlink, which is the title of Exercise 2
5. Follow the instructions in the exercise

Password Issues/Exploits

In this section, we'll take a quick look at some of the more common attacks against Unix passwords. We'll also talk in some detail about how passwords are stored and Unix authentication is performed in order to motivate our discussion of off-line, dictionary-based attacks.

The Bottom Line

- Attacks succeed because Unix passwords are *reusable*
- Non-reusable authentication technology exists today
- Not widely deployed due to admin/integration cost

The fundamental problem with Unix passwords is that they are reusable; the user or administrator always types the same password every time they want to log in or become root. If the attacker is able to steal a password, then they can always use that password at some point in the future until that password is changed.

Alternative technologies for replacing reusable passwords have been available for decades. Appendix A in this course book covers some of them in more detail: One-time passwords, public-key-based authentication, and Kerberos.

However, most organizations today still do not widely deploy these technologies, primarily because they are not incorporated by default into the systems that they purchase from their vendors (though this is slowly changing). The cost of acquiring, integrating, and maintaining a third-party security solution is usually too much for an organization to bear—especially if that organization is not particularly security conscious to begin with (a state that sadly describes the vast majority of IT organizations today).

Just as an FYI, here's an article describing how to integrate Google's two-factor authentication solution into SSH: *http://xmodulo.com/two-factor-authentication-ssh-login-linux.html*. It's not a complete enterprise solution, but it can be a free, quick way to get basic two-factor auth support for accessing critical servers.

Password Threats

- Exhaustive guessing
- Passwords written down
- "Shoulder surfing"
- Sniffing
- Social engineering
- Off-line attacks

The list of password threats goes on and on, and lots of time and energy have been devoted to figuring out how to capture, steal, crack, and otherwise obtain passwords. The first four items are generally well known, so we'll mention them in passing here and not waste time delving into these in detail:

- Some attackers will try *exhaustive guessing* attacks where they (manually or via a program) will simply try connecting to the system over and over again, trying different passwords each time. These attacks are slow because the attacker will typically be disconnected after 3-5 failed attempts and very noticeable to any administrator who observes the "repeated login failure" messages in their logs.
- Users and administrators will sometimes write down passwords that can then be easily stolen by an attacker.
- *Shoulder surfing* is observing somebody in the act of typing their password (or reading the password over their shoulder if the password is written down).
- *Sniffing* passwords is often possible because many networks (such as Ethernet) use shared media that potentially allows any host on the network to observe all traffic that passes by. The attacker simply runs a special sniffer program that watches the network for certain triggers (like the "`word:`" in a password prompt) and captures passwords and other sensitive data.

Social Engineering

```
From root@domain.com Sun Oct 11 18:12 PDT 1994
To: user@domain.com (J. User)
Subject: [IMPORTANT] Please change your password!
Date: Sun, 11 Oct 1994 18:23:34 -0700
From: System Administrators <root@domain.com>

We have become aware that some accounts on our
systems may have been compromised. Please change
your password to "NewWord" as soon as possible so
that we can determine the extent of the damage.
-- Your System Administration Team
```


Unfortunately, it is far too easy to forge email (and we'll show you how later in the curriculum, so don't worry). Furthermore, many mail servers will add the local domain name to email that comes in "unqualified" (that is, without any domain name portion at all). The result is a forged email like the one you see above.

On the face of it, the request in the email is ridiculous. Administrators should never have to request users to change their passwords in order for the administrators to investigate a security incident. After all, somebody with superuser access can access any file and become any user they wish. The problem is that your users may not know this and may follow the instructions of an email that appear to come from some legitimate authority. It is important that you train your users how to (a) detect fraudulent email like this and most importantly (b) bring it to your attention when they receive it so that you can track down the perpetrator and/or warn other users.

Of course, the modern equivalent of this email is all of the various sophisticated "phishing" scams that have been developed over the last few years. In this case, the attacker is usually interested in obtaining credentials, financial data, and/or other personally identifying information from your users, usually by getting your users to install a Trojan horse in the guise of an important security update, or by opening a malicious document or other attachment.

Off-line Attacks

1. Attacker takes hashed passwords from `/etc/shadow`
2. Attacker exhaustively tries all possible password strings

- Happily, even with weak DES56 hashing algorithm, there are about 7 quadrillion possible passwords (and 4096 salts)
- Desktop machines can test millions of strings per second

Off-line attacks require the attacker to steal the encrypted password strings for one (or possibly all) of your users. These encrypted password strings are made using a *one-way hash function* that makes it computationally infeasible for the attacker to decode the password. But the attacker can simply encrypt random strings until they get back an encrypted password that matches a password they managed to steal. Since the attacker knows the cleartext string they encrypted, they know the user's password.

It turns out that Unix passwords hashed with the older DES56 algorithm are silently truncated at 8 characters—that is, the password "`12345678`" and the password "`123456789`" are actually the "same" password as far as most Unix implementations are concerned. This means that the attacker only has to try strings of up to 8 characters.

The bad news for our attacker is that every Unix implementation allows the user to use any printable character (some even allow non-printable characters like control sequences, etc.)—there are about 95 possible characters. Even if the system forces users to use passwords that are 6-8 characters long, this is still about 7 quadrillion possible passwords (95^6 + 95^7 + 95^8 = 6.7e+15). Remember also that while a single password is encrypted with a single salt value, an entire file of passwords is encrypted with a random selection of salts—an attacker must encrypt a given string with any of the 4096 (64^2) possible salts to check it against all passwords in the file (assuming all salts are used).

Even with "ultrafast" versions of `crypt()` distributed with standard password-cracking tools, desktop machines are only up to doing low millions of DES encryptions per second, so such an attack takes a while. However …

Dictionary Attacks

Word Type	Search Size	Hit Percent
Usernames	130	2.7%
Machine Names	9018	1.0%
Names	7194	4.9%
Phrases	933	1.8%
/usr/dict/words	19683	7.4%
TOTALS:	36958	17.8%

In 1990, Dan Klein described some research he conducted on a sample set of about 14,000 accounts from a variety of systems. He created various "dictionaries" of common and uncommon words (aside from the groups listed here, he used dictionaries of science-fiction-related terms, astronomical bodies, famous people, Biblical references, etc.). The surprising result is how successful he was using a relatively small number of words and phrases.

In particular, people show an annoying tendency to use the account username, machine name, their own or some other common name ("Susan" turned out to be a big winner for some reason), or a word found in the Unix dictionary file `/usr/dict/words`. Klein's password guessing program was smart enough to try a few common permutations of certain letters (zeroes instead of Os, etc.) but very often got hits without having to resort to this effort.

In fact, as the above table demonstrates, a very small number of words were required to match almost 20% of the passwords that Klein sampled. On modern machines, these dictionaries could be tested (for a single salt value) in about a second (and a smart attacker might pre-compute these dictionaries for all possible salts and put the results in a database for later use—see, *http://project-rainbowcrack.com/*). Only one account is required to break into a system…

Here's the reference for Dan's original paper: Klein, Daniel V., "Foiling the Cracker: A Survey of, and Improvements to, Password Security", *USENIX Security Workshop Proceedings*, 1990 (available from *http://www.klein.com/dvk/publications/*). You can find the dictionaries that Dan used for his research (along with a bunch of other password-cracking dictionaries) online: *ftp://ftp.cerias.purdue.edu/pub/dict/dictionaries/*

Use Stronger Crypto

- DES56 too weak to stand up to modern computing power
- Better algorithms exist (Blowfish, SHA-1, MD5)
- Open Source Unix variants use stronger encryption

DES56 was originally chosen because it was computationally difficult for the machines that were available at the time (the early 1970s). Nowadays, of course, machines are immeasurably faster. In fact, machines are so much faster that the DES algorithm itself has been successfully broken (in less than three days using $250,000 worth of equipment—go to, *www.eff.org* and search for "DEScracker").

The good news is that much better algorithms have been developed. The Open Source Unix implementations (*BSD and Linux) allow the administrator to use better one-way hash functions for encrypting passwords—Blowfish under OpenBSD, MD5 under Linux, etc. Not only does encrypting strings with these algorithms take much more CPU time, but the algorithms also generally accept much longer passwords than the 8 character DES56 limit. Both of these factors mean that "dictionary"-type attacks take much longer.

Sadly, the commercial Unix variants haven't kept up, though this is slowly changing (Solaris 9, for example, introduced the option of turning on MD5 or Blowfish-style password hashes). This becomes a problem if you're managing a heterogeneous network of systems—you often end up having to fall back to the "lowest common denominator" encryption algorithm (i.e., DES56) if you want to maintain only a single password file format.

Additional Defenses

- User education
- Minimum password lengths
- Password expiration/aging/history
- Force good passwords at change time
- Encryption (switching to some extent)

Educating users on how to choose good passwords and how to protect their passwords must be the first line of defense. User education also helps prevent successful social-engineering attacks.

Other common password security mechanisms force users to change their passwords regularly (*password expiration*) under the assumption that the user will be forced to change their password before the attacker can decrypt and reuse it. However, when users are forced to change their password, they often try to "change" it to exactly the same thing. While most expiration systems prevent this, the user can change their password and then immediately change it back to their old password unless the system implements *password aging* (the user is unable to change passwords for some "cool down" period after the last change) or *password history* (the system keeps track of the user's last 5-10 passwords and doesn't allow the user to reuse one of them).

Most Unix implementations these days force users to choose passwords of at least six characters. Some systems allow the administrator to set other guidelines for user's passwords (must contain a case change, must contain a number or non-letter character, may not be a word in `/usr/dict/words`, etc.).

Full end-to-end encryption not only blocks sniffing attacks but also prevents session hijacking attacks as well. Using switched networks is not a panacea here because attackers now have tools that can overwhelm switches and force them to fall back to "hub" mode where all traffic is visible on all ports of the switch. Also, the attacker could spoof the MAC address of the default router and intercept all traffic leaving the LAN, or install a packet sniffer on a critical mail server or other machines where users regularly have to type their passwords for authentication, thus allowing the attacker to see all traffic destined to that machine.

Account Expiration, Etc.

- Set default values in `/etc/login.defs`, `/etc/default/useradd`
- Defaults only apply to new accounts created with `useradd`
- Manually set values with `chage`

Most Unix systems these days allow you to force users to change passwords on regular intervals (every 30, 45, 60 days, etc.). Typically, you can at least set how long a password is valid for (the "max" parameter), how long the user has to wait after changing their password before being able to change it again ("min"), and how many days in advance the system starts warning users that their password is going to expire ("warn").

Under Red Hat, the corresponding values are `PASS_MAX_DAYS`, `PASS_MIN_DAYS`, and `PASS_WARN_AGE` in `/etc/login.defs`. Red Hat also lets the admin set a number of days after the password has expired to lock the account completely—this is the `INACTIVE` parameter in `/etc/default/useradd`.

It must be stressed that these default settings are only applied to new accounts which are created with the system `useradd` command. If you're creating accounts in some other way, or if you want to set these values for existing accounts, you'll need to manually set these parameters with the Linux `chage` command.

Other Password Enforcement

For most Linux distros, PAM modules handle issues like:

- Dictionary checks
- Password "strength" requirements
- Password history

Linux distros support functionality for checking new user passwords against a dictionary, requiring some minimum number of non-alphanumeric characters, and keeping a "history" of user passwords to prevent repeats. Originally, these chores were handled by the `pam_cracklib` and `pam_unix` modules in the system PAM configuration, and actually, these modules still work. However, newer Linux distros are now including `pam_pwquality` and `pam_pwhistory`, which are improved modules for handling the same tasks.

There's too much configuration detail to go into now, but I've written an article on the subject that you may find helpful: *http://www.deer-run.com/~hal/linux_passwords_pam.html*

Linux implements "lockout on failure" functionality via the `pam_tally2` PAM module—you can read the manual page for further details or consult the document at the Deer Run website referenced above. `pam_tally2` supports automatically re-enabling accounts after a certain timeout period or requiring manual admin intervention to unlock the account.

It should be noted, however, that Unix-like operating systems have historically been laggards in adopting these kinds of strong password controls. For example, this functionality only appeared in Solaris with Solaris 10 (for more Solaris details, read the comments in `/etc/default/password`).

Cleaning Up Accounts

Default "system" accounts with UID < 1,000

Many accounts are unused– delete for greater auditability?

- May impact software installs, operation
- May leave behind "unowned" files

Other accounts are necessary but should be blocked/disabled to prevent logins

Unix systems typically ship with a number of default accounts that are not associated with actual users. Sometimes, these accounts are associated with system processes—like the `sys` and `lp` accounts that are used by the system accounting and Unix printing services, respectively. Sometimes these accounts are associated with specific applications—`ftp` for anonymous ftp, www for Web servers, etc.—and sometimes the accounts are just historical, like the `uucp` accounts that may be present even though very few sites still use UUCP for email/news. The Linux convention is that these "system" accounts have UIDs less than 1,000. The exception is the `nobody/noaccess/ fsnobody`accounts that have UIDs larger than 60,000 (the maximum 16-bit UID is 65,535).

From an auditability perspective, it's a good idea to remove as many of these accounts from your password file as possible—if the password file is shorter, you're more likely to detect when somebody adds a new account. Also, removing these accounts means that attackers can't "reactivate" the account and use it as a backdoor to access your system, and again because the file is shorter, you're more likely to notice if one of the remaining accounts has been reactivated. You can use `userdel` on Linux to delete accounts.

However, sometimes it's impossible to remove an account because critical applications won't run without the account entry in `/etc/passwd`. Also, many automated software installs expect accounts like `bin` and `daemon` to exist in the password file and will die when they try to do commands like `chown bin …`

Another problem is that some of the accounts you delete may actually own files and devices on the system (for example, the Solaris `uucp` accounts owns some of the `tty` devices and programs like `tip` and `cu`). If you end up assigning these UIDs to new user accounts, then those users may end up with "extra" privileges on the system that they shouldn't have. For this reason, some sites have a policy of never deleting accounts from the `passwd` file so that they never accidentally reassign a UID.

If you're more in a mood to block accounts, it's generally considered good practice to both change the user's entry in `/etc/shadow` to an invalid hash value as well as setting their login shell to something that isn't a valid shell, such as `/dev/null` or `/bin/false`. On Linux systems, "`usermod -L -s /dev/null` <*user*>" will simultaneously lock the user's `/etc/shadow` entry (the Linux standard is to put a "!" at the front of the password hash, rendering it invalid but allowing you to recover the original hash if you decide to let the user back in) as well as set their login shell to `/dev/null`. And because there are always multiple ways to do things in Linux, you could accomplish the same mission with the "`passwd -l` <*user*>" and "`chsh -s /dev/null` <*user*>" commands as well.

You can also expire an account with "`chage -E 0` <*user*>". This is another method of preventing logins as a particular user and may be used in combination with locking the password hash and setting an invalid shell.

Default User Environment

- Set `UMASK` and `PATH` in `/etc/profile` and `/etc/.login`
- Tweak default files in `/etc/skel` used by `useradd` prog
- Can always be overridden by users

If a user logs in who does not have any start-up files in their home directory, the system consults either the `/etc/profile` (`sh` and `ksh`) or `/etc/.login` (`csh`) files. You may wish to set some sensible values for `UMASK` and `PATH` in these files. Note that the `useradd` program copies initial start-up files from `/etc/skel` and puts them in the new user's home directory. Again, you may want to set appropriate parameters in these files as well.

Of course, users can later edit their dot files and change the settings, but if you choose good settings, you'll find that most users don't bother.

Other Tweaks

- Block FTP access by listing users in `/etc/ftpusers` file
- Control access to `crontab` command with `cron.allow` and `cron.deny`

If you're still running FTP for some reason (e.g., the machine is an anonymous FTP server), you should create an `/etc/ftpusers` file. Even if you're not running an FTP daemon currently, creating this file is a good idea just in case you ever do end up starting an FTP daemon on the machine.

The name of the `/etc/ftpusers` file is exactly backwards: The file lists those users who should **never** be allowed to FTP into the system. The superuser must always be in this file (along with any non-user accounts such as `bin` or `nobody` that you left in the password file), but feel free to add any unprivileged users who shouldn't be moving files via FTP. In many cases, the FTP server is *only* for the use of anonymous FTP users (normal users would use SSH to access the system) so *all* users in `/etc/passwd` might be listed in `/etc/ftpusers`. Be careful not to add the `ftp` account to the `ftpusers` file, though, as that would prevent anonymous access.

The `cron.allow` file lists the users who are allowed to run the `crontab` command to modify/add/delete `cron` jobs (`at.allow` does the same thing for the `at` command)—all other users are restricted from running this command. Even if a user is not listed in `cron.allow`, `cron` jobs can still run as that user but will need to be maintained by the system admin. For "hardened" machines or production systems, probably only root needs to be in `cron.allow`/`at.allow`.

Note that if `cron.allow`/`at.allow` don't exist, then the system will look at `cron.deny`/`at.deny` for a list of users who are not allowed to use these commands. Any user not explicitly listed in the `*.deny` files would be allowed to use the commands. On non-production systems, you can use `{cron,at}.deny` to deny `crontab` access to individual users while letting everybody else use the `crontab` and `at` commands by default. On production systems, however, you should just delete the `*.deny` files and stick with the `*.allow` files.

Red Hat puts the `cron.*` and `at.*` files in `/etc`, but there is wide variation across Unix-like operating systems. The easiest way to find these files for any particular flavor of Unix/Linux is to look at the manual page for the `crontab` command ("`man crontab`"). At the end of the manual page should be a section that lists the configuration files for the `crontab` command, including the `*.{allow,deny}` files.

As long as we're talking about cron security issues, it's worth noting that the `crontab` command itself is set-UID to root and ends up reading and writing files to and from the crontab directories as the root user. Thus, it is possible to make all of the crontab files owned by root and only readable by the root user. This can help hide information about programs being run via `cron` from local attackers who have not yet gained privileged access to the system.

Red Hat has `cron` information spread out in both `/var/spool/cron` and the `/etc` directory, so it's actually a bit of work to set the most restrictive permissions on all relevant files:

```
chown -R root:root /etc/cron* /var/spool/cron
chmod -R go-rwx /etc/cron* /var/spool/cron
```

Again, consult your vendor documentation (or the CIS Benchmark Guide) for the location of `cron` entries for your operating system.

Root Logins on Console Only!

- Restrict root logins to console device via `/etc/securetty`
- Also set "`PermitRootLogin no`" in `sshd_config`!

Experts agree that the best policy is to force users to log in under an unprivileged account and then use `su` or `sudo` to get root access. This provides maximum audit trail and also forces attackers to compromise two accounts to "get to root."

Of course, in an emergency, it may be necessary to log in as root on the system console to save the system. However, root logins should be restricted to this console device only (and ideally the console is locked up in a secure data center). Under Red Hat, `/etc/securetty` contains all of the devices where root logins are allowed. By default, this file ships with a number of devices and virtual consoles listed, but you could get by with just listing the main system console (`tty1`).

As we discussed earlier, SSH daemons have a `PermitRootLogin` parameter that says whether root SSH logins are allowed (over and above the settings in system files like `/etc/default/login` or `/etc/securetty`). Again, make sure this parameter is set to "`no`"–Red Hat and many other Linux distros ship with this value set to "`yes`" by default.

Similarly, other applications (like GUI login screens) may have specific configuration options that control whether or not root logins are permitted. Unfortunately, you may end up having to check the documentation for each individual login application. This is a good argument for strictly limiting the number of ways users can log into your systems—less configuration to maintain!

Lab Exercise

- Password cracking!

- Have fun with John the Ripper
- Win valuable prizes?

Exercises are in HTML files in two places: On the course USB media and in the home directory of the SANS user in the lab virtual machine. These two locations contain identical copies of the exercise files—it just depends on whether you prefer to look at them from a web browser inside the lab VM (because you're working in full-screen mode) or from your host operating system.

1. Open your web browser
2. Select "File… Open…" from the browser menu (Ctrl-O is the usual keyboard shortcut)
3. This is Day 2, Exercise 3, so navigate to *.../Exercises/Day_2/index.html* and open this HTML file
4. Click on the hyperlink, which is the title of Exercise 3
5. Follow the instructions in the exercise

Sudo

As long as I'm bringing up root-access-control issues, let's talk a little bit about the sudo utility being a better way of doling out root access.

One of the common problems in Unix security is the "all or nothing" security model that Unix enforces—once you give a user the root password on a machine to perform one operation (like mounting floppy disks or CD-ROMs), they can use that access to do anything. Sudo is a mechanism for giving away privilege in a controlled fashion and keeping a good audit trail of what people are doing with those privileges.

Sudo was originally developed at the University of Colorado Boulder, and popularized because two of those developers (Garth Snyder and Trent Hein) were co-authors with Evi Nemeth (a faculty member at Colorado-Boulder) on *The Unix System Administration Handbook*, which includes a chapter on sudo.

What Does It Do?

- Allows the user to run individual commands with privilege
- User authenticates *with their own password*, rather than root password
- Administrator controls which commands any given user may execute via sudo
- Every command is logged, configurable alerts

The basic idea is that sudo allows the administrator to define a limited set of commands that a given user is allowed to run with privilege—usually as root, although sudo allows the administrator to define other alternate users that the commands will run as. Each time the user uses sudo to execute a command, logging data is produced so that the administrator has a complete audit trail in case the user makes a mistake.

When the user wants to run a command with privilege, they execute that command via the `sudo` program (e.g., "`sudo cat /etc/shadow`"). Sudo prompts the user for their own password to verify the user's identity. *The user never needs to know the superuser password for the system*.

Sudo was not the first, nor is it the only program to implement this idea. You can find a list of other similar programs at:

`http://www.sudo.ws/sudo/other.html`

Sudo is, however, probably the most widely used tool of this type, so the people you hire are more likely to be familiar with sudo already as opposed to some of the other tools.

What Problems Does It Solve?

- *Least privilege:* Give operators and DBAs only the commands they need
- *Accountability:* Check the sudo logs to see who crashed the system
- *Termination:* Don't have to change the root passwords when somebody leaves

So, we can see right away that sudo solves at least two critical problems. The first is the granting of *least privilege*, which is giving users as close to the minimum amount of access they need to get their work done. Sudo also provides *accountability* through a detailed audit trail of all commands the user executes (though we'll talk about ways users attempt to circumvent this).

What's less obvious until you've been using sudo for a while is that sudo makes life much easier for system administrators when users leave the company. If you give out root passwords to your users, chances are you end up having to change root passwords on your systems whenever those users leave your organization (some sites don't bother to change passwords when somebody leaves, but that's a different problem). On a large network, this is a painful, time-consuming process. However, if you give people root access via sudo, then all you have to do is just delete their entry from a central sudo configuration file, which is a huge win.

In fact, at some sites I've worked at, the more junior system administrators don't even know the root passwords for the systems they're maintaining. They just use sudo for everything. Obviously, there are some system admins who know the actual root passwords (in case sudo breaks down for some reason), and you'll still have to change passwords when these people leave. It's worth noting that some Linux distributions (e.g., Ubuntu) now run entirely with sudo and don't even set a reusable root password for the system.

Overview of Features

- Centralized, flexible configuration file (or LDAP or NIS)
- Support for many, many different authentication types
- Timed sessions limit password retyping
- Optional humorous error messages

Sudo is designed so that you can (and should) have a single configuration file that is copied to all of the systems in your network. This configuration file allows you to specify individual access control rules on a per-machine or per-network basis (we'll be looking at a sample configuration file in detail in just a few slides).

While having a local configuration file on each machine is the normal practice, sudo also supports using LDAP or NIS from a central server to export configuration rules. I've even seen sites that keep their `sudoers` file in a central NFS-mounted directory (like `/usr/local`) rather than push copies of the file to each machine. The problem with all of these approaches, however, is that a network failure means that sudo stops working, and you may need your sudo access in order to resolve the networking issue. It is best to have a local copy of the `sudoers` file on each machine.

If the user had to enter their password every time they ran a command via sudo, it would be a huge hassle. So, what sudo actually does is prompt the user for their password only on the first command. As long as the user keeps using sudo to execute commands, sudo "remembers" that the user entered their password correctly and doesn't prompt again. If the user hasn't run a command via sudo in a certain interval (the default is five minutes), sudo will prompt them for their password again (this is to protect people who walk away from their workstation and leave their screen unlocked). By the way, we've been saying "password" throughout this discussion, but sudo actually supports many other forms of authentication, such as Kerberos, S/Key and OPIE, and SecurID cards.

The original sudo developers also had a sense of humor, so you may optionally enable humorous, insulting error messages ("Life just hasn't been the same since the electroshock treatments, eh?") instead of boring old "permission denied" messages. Sometimes these little bits of humor actually make introducing sudo more palatable at certain sites. Note that the insulting error messages need to be enabled during compilation (as well as in the sudo configuration file), and many of the pre-compiled versions of sudo that ship with various operating systems were compiled without insult support. Those pesky Linux/Unix vendors have no sense of humor …

Usage Example

```
% sudo cp /etc/passwd /etc/passwd.old
We trust you have received the usual lecture from
the local System Administrator. It usually boils
down to these two things:

        #1) Respect the privacy of others.
        #2) Think before you type.

Password:
% sudo cp /etc/shadow /etc/shadow.old
% sleep 300
% sudo rm /etc/{passwd,shadow}.old
Password:
```


Here's some sample output from a sudo session.

The first time you run sudo on a machine, you get a mini-lecture from the program. This lecture can be selectively disabled in the sudo configuration file, and/or you can provide your own lecture text in a separate configuration file.

Sudo then prompts the user for their password—remember, this is the user's password, not the root password. Assuming the user gets the password right, the command is executed (in this case making a backup copy of `/etc/shadow` called `/etc/shadow.old`).

The next time a user runs a command via sudo, it "remembers" that the user has already typed their password and doesn't prompt the user to re-enter it. However, if the user doesn't enter another command via sudo for five minutes, then sudo will re-prompt them for their password the next time the `sudo` command is run.

At this point, you may be wondering how sudo knows the last time a user ran a command via `sudo`. On each machine, the `sudo` command will create a timestamped file for each user/pty combo in a private directory—usually `/var/run/sudo`, though this varies from OS to OS. When the `sudo` command is executed, it simply checks to make sure the file for the user isn't older than five minutes, and then updates the timestamp on the file to indicate the current time.

How Users Thwart sudo

Use sudo to invoke a shell:

```
% sudo /bin/sh
# ^D
%
```

Use a shell escape in another program:

```
% sudo vi
   :shell
#
```

SANS | SEC506 | Securing Linux/Unix 158

Sudo's audit trail only works if the users religiously run the `sudo` program to execute all commands. However, it's a common tendency for frustrated users to simply run "`sudo /bin/sh`" or "`sudo /bin/su`" in order to get a root shell. At this point, any commands they type will not be logged by sudo.

Some admins try and prevent this problem by configuring sudo to block users running these commands (though as we'll see later, this can be circumvented). Unfortunately, there are a number of standard Unix commands (the `vi` editor for one) that allow the user to do a "shell escape" and get an interactive shell—this shell has the privileges of the user the process was running as. In our example above, the user runs "`sudo vi`" to get a root `vi` window and then runs the `:shell` command in `vi` to escape to a root shell.

As we move through the configuration examples, I'll point out some functionality built into sudo to try and prevent these sorts of escapes. Ultimately, however, there isn't a perfect solution to stop all such possible privilege escalations, so you (a) have to trust your users to do the right thing, and (b) monitor your sudo audit trail to make sure this trust is not misplaced. Trying to define a sudo policy that covers all of your bases is generally impossible and makes your sudo configuration too difficult to maintain.

Modifying `/etc/sudoers`

- `/etc/sudoers` is just an editable text file
- However, syntax errors break `sudo`
- Use `visudo` which does syntax checking before saving
- Use a version-control system!

Before we get to the syntax of the `/etc/sudoers` file—the main configuration file for sudo—it's worth talking about how you should maintain this file. The `sudoers` is just a text file and can be edited using any normal Unix text editor. However, if there are any syntax errors in the file, `sudo` generally stops working for *everybody*. Also, you need to be careful that multiple administrators don't try to modify the `sudoers` file at the same time and corrupt the file.

The solution is to use the `visudo` command (analogous to `vipw`) for editing the `sudoers` file. First, `visudo` takes an exclusive lock on the `sudoers` file so that only one person can be running `visudo` at any given moment. After the administrator makes their changes via the editor run by `visudo`, the `visudo` program syntax-checks the new `sudoers` file. If there are any problems, `visudo` prompts the admin to either re-edit the file or abort without making an update (there's also an option to save the file anyway, but don't ever do this).

Even if you are using `visudo` for file locking, you should also use RCS (or some other revision control system) to keep a revision history of the `sudoers` file (and any other critical system configuration files). You never know when you'll want to go back to an older copy of this file.

/etc/sudoers Overview

- Define groups of users, commands, hosts
- Define defaults (global, per machine/user, per group)
- Use groups and literals to define rules
- Rules say which user can run which commands on which hosts as what user

`sudoers` files are generally all written the same way. The first part of the file defines groups of machines, users, and commands and gives these groups names that will be used in the rest of the `sudoers` file. The next section of the file defines certain defaults—typically global defaults, but as we'll see, you can override the global defaults for certain users or certain machines. The remainder of the file is made up of rules saying which users are allowed to run which commands on which machines. We'll look at all of these sections in detail in the next several slides.

Remember that the goal is to have a single configuration file that can be maintained in one place and then pushed out (via `rsync` or some other mechanism) to all of the machines in your network. This makes administration much easier.

Defining Users

- Could create `User_Alias` entries:

```
User_Alias      ADMINS= bob, hal, mary
User_Alias      OPER= jane, john, ramesh
User_Alias      DBA= pei, zane
```

- But maintaining user lists is a pain
- Better to use `%<group>` syntax

The `User_Alias` syntax is designed to allow administrators to create macros that define particular communities of users—system admins, DBAs, normal users, etc. The problem with `User_Alias` is that every time you add or remove a user, you also have to make the corresponding change to the `User_Alias` lists in the `sudoers` file.

The good news is that sudo also supports the `%<group>` syntax, which allows you to specify the "user" portion of sudo rules using Unix group names, rather than maintaining your own `User_Alias` entries. The advantage here is that now users automatically inherit sudo privileges based on the group(s) you assign them to when their account is created. Sudo will look at both the user's primary group as defined in their `/etc/password` entry as well as any groups the user has been assigned to in `/etc/group` and give that user all of the privileges for all of the groups of which they are a member.

So, use `%<group>` instead of `User_Alias`, because it makes administering the `sudoers` file much less work and much less prone to error.

Hosts and Commands

```
# Groups of machines defined here
Host_Alias     DEV_LAN=10.1.0.0/255.255.240.0
Host_Alias     DBSERV=castor,pollux
Host_Alias     SQA=192.168.0.0/255.255.0.0

# Command aliases
Cmnd_Alias  SHUT = /usr/sbin/shutdown, \
                   /usr/sbin/halt,/usr/sbin/reboot
Cmnd_Alias  PROCESS=/bin/kill,/bin/pkill,/bin/nice, \
                   /bin/renice,/usr/bin/systemctl
Cmnd_Alias  SHELLS=/bin/*sh, /bin/su
```


Here is part of a sample `sudoers` file. Note that comments and blank lines are allowed. Continuation lines with backslash ("\") are also allowed, as you can see.

First, we define special groups of machines using the `Host_Alias` configuration lines. You can use either hostnames or IP addresses or address/mask combinations or some combination of all three. In our example, the networks that the software developers live on in our network (`DEV_LAN`) is the IP address range `10.1.0.0` through `10.1.15.255`—we express this using the netmask shown above. We also have two database servers (`DBSERV`) whose hostnames are `castor` and `pollux`. Finally, we define the range of addresses for our release engineering/quality assurance labs (`SQA`), which are all numbered somewhere in network `192.168`.

Then we define some collections of commands using the `Cmnd_Alias` configuration directives. We're going to define some rules that allow our operations staff to shut down machines (the `SHUT` command set) and control processes (the `PROCESS` command set).

We are going to be using the `SHELLS` list to create a rule that attempts to prevent people from doing things like "`sudo /bin/sh`" (notice also that we're including `/bin/su` because "`sudo su`" is equivalent to "`sudo sh`"). There are lots of command shells available in the typical Unix/Linux environment, and the list varies widely from OS to OS (not to mention extra shells that may have been installed on a per-site basis), so make sure that the list in your central `sudoers` file contains all possible shells you will be running at your site (this can be a big list). Notice that shell wildcards are allowed in rules in the `sudoers` file.

Some Useful Defaults

```
Defaults  syslog=local2, !mail_no_user
Defaults  !lecture
Defaults  password_timeout=3
Defaults  !visiblepw

Defaults:%users     noexec
Defaults:%users     lecture
Defaults:%users     lecture_file=/etc/sudolect

Defaults@DEV_LAN         password_timeout=5
```


This slide shows some of the interesting options you can set in the `sudoers` file, but there are literally dozens of other customization options available (see the `sudoers` manual page). As you can see, you can have multiple `Defaults` entries in the `sudoers` file, although we also could have expressed the first four lines above as a single very long `Defaults` entry. The method used above is more readable, though.

It's a good idea to send all of your sudo logs to a central Syslog server for easier auditing. However, you probably want to keep your sudo logs separate from your other logging streams to make it easier to audit the sudo log trail. So, using a Syslog facility like `LOG_LOCAL2` (this is the default for sudo, btw) is probably a good idea. Note that sudo is also capable of writing its own audit log file independent of the Syslog system (see the `logfile` option in the `sudoers` manual page), but since Syslog is capable of writing local log files in addition to logging things to a remote server, there doesn't seem to be much point in having sudo also create a local log file.

Sudo can also send real-time alerts via email, but at a large site, this gets very annoying very quickly. Most of the email alerts are disabled by default, but the `mail_no_user` alert—when a user tries to use sudo but has no privileges defined in the `sudoers` file—is on by default. The "`!mail_no_user`" syntax tells sudo to disable this option.

If you use sudo a lot, getting the new user "`lecture`" from sudo the first time you run sudo on a machine gets old in a hurry—so "`!lecture`" to disable the lectures by default. You can also change the `password_timeout` interval to something other than the default five-minute timeout.

The `!visiblepw` option prevents users from accidentally exposing their passwords when running commands like "`ssh` *`somehost`* `sudo` *`somecommand`*". The problem here is that the SSH client will typically not be allocated a pty for these kinds of remote commands. But without a pty, sudo can't disable echo when the user types their password. Thus, the user's password would end up being visible in the window where they're running

the SSH client, which is a Bad Thing™. !visiblepw tells sudo not to execute if it is unable to protect the user's password from exposure. Note that this is the default for sudo, so here we are just reinforcing that default.

After you've defined some reasonable global options, you can override those options (or set additional options) for different groups of users and hosts. Notice the slight difference in syntax for per-user defaults (Defaults:*USER* with a colon) and per-machine defaults (Defaults@*MACHINE*, with an "@").

Earlier, we talked about how users will often try to escape from sudo by invoking shell escapes in various programs. The noexec option tries to prevent this by inserting a bogus shared library into the LD_PRELOAD search path of commands invoked via sudo: The bogus shared library has a deliberately broken exec() call that should prevent shell escapes (note that there were problems with the initial noexec implementation not working on AIX—if you're using AIX, make sure you have a fairly recent version of sudo).

Be a little careful with noexec, however, because it can have negative effects on useful programs, too. For example, both the visudo and "crontab -e" commands invoke the user's text editor via an exec() call. These commands will not work if you set noexec. noexec is also a problem when users want to "sudo bash" to spawn a shell. The shell will start, but the user won't be able to run commands because the shell can't exec() them! In our case, we don't want our regular %users to spawn interactive shells, so this is actually a feature from our perspective.

We want our normal %users to receive the sudo lecture we disabled globally in the previous set of defaults. As you can see, sudo even lets you create your own lecture message and put it into a text file. Of course, you will have to have a copy of this file on every system—if the file is missing, sudo automatically reverts to the standard lecture message that's compiled into the program.

The last configuration directive shows us overriding the password_timeout interval on the development machines to make things a little more user-friendly in this less secure environment.

/etc/sudoers – Rules

```
%prodadmin      SQA = ALL
%sysadmin       ALL,!SQA = ALL

%oper           ALL = SHUT, PROCESS
%users          DEV_LAN = ALL, !SHELLS

%dba DBSERV = (oracle) ALL
%dba DBSERV = (root) /etc/init.d/oracle
%dba ALL = (root) sudoedit /var/oracle/tnsnames.ora
```


Finally, we define some specific rules for various users and machines. The basic rule syntax is:

<userlist> *<hostlist>* = *(<runas>)* *<cmdlist>*

In our examples, *<userlist>* is always a *%group* type entry, but you could use a pre-defined `User_Alias` or even individual usernames. *<hostlist>* and *<cmdlist>* are lists of hostnames/commands or aliases defined with `Host_Alias` or `Cmnd_Alias`.

The optional *(<runas>)* field is a list of usernames that the user is able to run the command as (root is the default if no *<runas>* entry exists)—`sudoers` also has a `Runas_Alias` declaration for defining *<runas>* groups, but this is rarely used. Note that if you want to use sudo to run a command as a non-root user, you use the syntax "`sudo -u` *username …*".

The wildcard `ALL` can be used anywhere. You can also create exceptions to a list using the "`!`" symbol (see the second line above which says "`ALL,!SQA`"—all machines *except* the machines on the testing networks).

So, the first line in the example above says the users in the production sys admin group can run `ALL` commands on any machines on the networks defined by the `SQA` host alias. The second line says that the regular sys admins can run `ALL` commands on `ALL` machines *except* the `SQA` machines. Thus, each administrative group is master of their own machines but has no access to other machines.

The operators are allowed to reboot/shutdown and manage processes on any machine in the company.

Then we allow normal users on the development networks to execute any command they want via `sudo` *except* the standard Unix shells. In other words, the software engineers all have root privilege, but only via the `sudo` command.

Finally, we define a security policy for our `DBA`s. They can run any command they want as user `oracle`, but only on the database server machines—this is how you stop your DBAs from spending their entire lives logged

in as the `oracle` user. You might also want to set the `runas_default` option for the `%dba` group to be `oracle` so they don't have to type "`sudo -u oracle ...`" all the time. See below:

```
Defaults:%dba      runas_default=oracle
```

The problem you're going to run into as soon as you start forcing users to use `sudo` to gain `oracle` privileges, however, is that the environment variables that are normally set from `~oracle/.profile` do not end up being set when you `sudo` to the `oracle` user. Check out the `env_keep` option in the `sudoers` man page to see how you can allow certain variables to migrate from the user's environment to the environment of the process being run by `sudo`.

The second DBA rule allows them to run the `/etc/init.d/oracle` startup script as root to cleanly start/stop the Oracle instance on the DB servers.

The last DBA rule allows them to use `sudoedit` (a variant of the regular `sudo` command) to safely edit a particular file (yes, you cannot only specify the commands a user is allowed to run but also limit them to specific command line arguments). `sudoedit` uses `sudo` to make a temporary copy of the file being edited that is owned by the user running the `sudoedit` command. `sudoedit` then invokes the user's preferred editor (specified via `$EDITOR` or `$VISUAL`) on the copy of the file. Once the user quits out of the editor, the original file gets overwritten with the user's changes. Basically, `sudoedit` is another way to help prevent users from doing shell escapes to get away from their sudo access restrictions.

You're Not as Secure as You Think

Try to prevent `sudo /bin/bash`:

```
%users DEV_LAN = ALL, !SHELLS
```

However, it doesn't entirely work:

```
% cp /bin/bash /tmp/bash
% chmod 755 /tmp/bash
% sudo /tmp/bash
#
```

SANS | SEC506 | Securing Linux/Unix 167

One of our rules on the previous slide attempted to allow users to run any command they want *except* the shell commands (and `su`) that would allow them to escape from sudo. It turns out, however, that these sorts of rules don't really work, because sudo grants/disallows access to certain commands based on path names. If the user copies `/bin/bash` to some other pathname, then they can execute their copy of the shell with root privilege as shown above. Clearly, you could (and probably should) parse your sudo audit logs and detect users running commands from non-standard directories via sudo.

One piece of good news is that the `noexec` option we set for the `%users` group breaks their shell functionality:

```
$ sudo /tmp/bash
# id
bash: /bin/id: Permission denied
# cd /tmp
# echo *
bash hsperfdata_root …
```

Commands like `cd` and `echo` work because they are shell built-ins. But any non-built-in command will fail due to `noexec`.

Ultimately you have to define some sensible defaults and trust your users not to try to circumvent them. Rather than doing "`ALL, !...`"-type exclusions, you could try to create an explicit list of commands you think your users might need to execute with root privileges, but there are so many possible commands that this would be pretty difficult—or at least a *very long* list.

One More sudo Problem

Output redirection happens without privilege:

```
% cd /etc
% sudo sed s/bash/zsh/ passwd >passwd.new
passwd.new: Permission denied
```

Workaround is to use `sudo bash -c ...`

```
% sudo bash -c
     'sed s/bash/zsh/ passwd >passwd.new'
```


Another classic user interface problem with `sudo` is that when you run a command and then try and redirect the output to a new file, the output redirection happens with the privileges of the user rather than with root privilege. One workaround is to simply redirect the output to a directory (like `/tmp`) where the user has write permission and then use `sudo mv ...` to put the file where you want it. Of course, writing sensitive files into a world-writable directory like `/tmp` is dangerous.

A better way to "fix" this problem is to wrapper the original command up inside a `sudo bash -c '...'` as we're doing on the slide. This way, the whole bash shell command-line happens with privilege, including the output redirection.

Another workaround is to pipe the output of the original command into `sudo tee`, `which` is a program that writes its input to a file and also writes the same input to the standard output (it's used in the middle of shell pipelines so you can save intermediate results in a file).

```
sudo sed s/bash/zsh/ passwd | sudo tee passwd.new >/dev/null
```

In this case, all we really want to do is create the file `passwd.new`—we don't want the normal output of `tee`, so we just redirect to `/dev/null`.

Summary Info – Sudo

Homepage

http://www.sudo.ws/sudo/sudo.html

Download From

http://www.sudo.ws/sudo/stable.html

Important Docs

http://www.sudo.ws/sudo/man/sudoers.html

Sudo is now maintained by Todd Miller of GratiSoft, Inc.

Lab Exercise

- Configuring and using Sudo

- Set up Sudo so you can use it for the rest of the week
- Experiment with known issues

Exercises are in HTML files in two places: On the course USB media and in the home directory of the SANS user in the lab virtual machine. These two locations contain identical copies of the exercise files—it just depends on whether you prefer to look at them from a web browser inside the lab VM (because you're working in full-screen mode) or from your host operating system.

1. Open your web browser
2. Select "File… Open…" from the browser menu (Ctrl-O is the usual keyboard shortcut)
3. This is Day 2, Exercise 4, so navigate to *.../Exercises/Day_2/index.html* and open this HTML file
4. Click on the hyperlink, which is the title of Exercise 4
5. Follow the instructions in the exercise

Warning Banners

This section examines presenting warning banners at login time for various login services. This may seem kind of silly, but in the United States the "Wiretap Act" (18 U.S.C. 2510-22) and the Electronic Communications Privacy Act (18 U.S.C. 2701-12) provide a "consent exception" for administrators to monitor system and network activity, provided the user has been shown a banner that informs them they have no reasonable expectation of privacy on the network. So, these banners can act as a "get out of jail free" card.

The Long and the Short of It

```
This system is for the use of authorized users only. Individuals
using this computer system without authority, or in excess of their
authority, are subject to having all of their activities on this
system monitored and recorded by system personnel.

In the course of monitoring individuals improperly using this
system, or in the course of system maintenance, the activities of
authorized users may also be monitored.

Anyone using this system expressly consents to such monitoring and
is advised that if such monitoring reveals possible evidence of
criminal activity, system personnel may provide the evidence of
such monitoring to law enforcement officials.
```


Here's a shorter message that covers the same idea:

Authorized uses only. All activity may be monitored and reported.

Note that the second word is "uses", not "users"—"uses" covers not only unauthorized users doing bad things, but also covers authorized users exceeding their authority.

The long message was originally developed by the U.S. Department of Justice. You should consult your legal counsel before using any such message. You should also talk to your local CISO or other security policy person because your site may have other requirements (regulatory or otherwise) about the language that can/should appear in these messages.

Where to Put Message?

- Need to display message *before* login:
  - `/etc/issue` for text-mode console logins
  - `Banner` option in `sshd_config`
- Putting message on login GUI a hassle on some OSes
- Don't forget legacy login services like FTP

Whatever message you decide on, the next question is how to display this message. The relevant US statutes require that the message be displayed *before* the user is given a chance to log in. That means the `/etc/motd` file is *not* appropriate because it only gets displayed *after* the user logs in.

`/etc/issue` is displayed before the login prompt on the system console and other physical tty devices (e.g., if you have a modem or terminal attached to one of the system serial ports). Similarly, setting the `Banner` option in your `sshd_config` file will display a warning message before SSH logins (`Banner` applies to v2 protocol sessions only). Note that the argument to the `Banner` option is the name of a file containing the warning message to be displayed, so you could just set this to `/etc/issue` to make things easier if you want.

However, there may be other login protocols on the system. For example, if you're still using legacy login services like Telnet, `rlogin`, or FTP, then you'll want to figure out some mechanism for displaying a banner here. Consult your vendor documentation—there's usually some way to get this done. For example, the `xinetd` process on Red Hat systems can be configured to display banners. Red Hat also has `/etc/issue.net` which is displayed before `telnet` logins.

Setting banners on those X Windows login widgets can also be a hassle sometimes. So, let's talk about that in some detail…

X Login Banners

- Nasty and different for each OS release
- Why isn't this easier?

On RHEL 5.x, things were simple. You added the `InfoMsg` file setting to the `[greeter]` section of your `/etc/gdm/custom.conf` file. Usually, the easiest thing to do was to just set `InfoMsgFile` to `/etc/issue` so you only had to maintain a single warning message file:

```
[greeter]
InfoMsgFile=/etc/issue
```

But that was too straightforward apparently, so RHEL 6.x forced you to edit `~gdm/.gconf/apps/gdm/simple-greeter/%gconf.xml` and add entries that look like this:

```
<entry name="banner_message_text" type="string">
<stringvalue>The text of your warning banner goes here.

You are allowed to have multi-line messages.</stringvalue>
</entry>
<entry name="banner_message_enable" type="bool" value="true"/>
```

Whereas the RHEL 5.x functionality was implemented as a "click-thru" type pop-up box, the text you put into %gconf.xml shows up on the login dialog box. This is also a step backward, IMHO.

RHEL 7.x is slightly better. Create the `/etc/dconf/db/gdm.d/01-warning-banner` with the following settings:

```
[org/gnome/login-screen]
banner-message-enable=true
banner-message-text='Your message here'
```

After you create this file, run "`dconf update`" to push the change into the gdm configuration. Note that the message text here will be displayed on the login window dialog, so you don't have room for lengthy text.

One shortcut that I've seen at some sites is encoding the warning banner into a GIF or JPEG image and using this image instead of the default vendor graphics on the login screens.

Kernel Tuning for Security

Unix systems generally allow administrators to set system limits and tweak network interface parameters via the system kernel. Some of these settings can be helpful from a security perspective.

Two Classes of Parameters

Network configuration parameters:

- Help prevent denial-of-service, spoofing
- Drop obviously bogus network traffic
- Possibly improve network performance

System resource limits:

- Help prevent denial-of-service
- Modify core dump behavior
- Other vendor-specific options

Generally, there are two different classes of parameters you can fiddle with in the system kernel that is useful from a security perspective.

One type of kernel parameter that's interesting to security folks is network configuration settings. Generally, these settings can be used to "harden" your network stack against various forms of abuse—from denial-of-service attacks to dropping obviously nasty traffic like source-routed packets. If you know what you're doing, tuning these parameters can also be used to improve network performance and throughput, but that's beyond the scope of this course.

You can also tune various OS limits to prevent certain trivial denial-of-service attacks. Your vendor may also allow you to do other interesting things like enabling "stack protection" to help stop buffer overflow attacks.

Vendors are also starting to provide hooks to either prevent core files from being dropped or at least control how and where they're dropped on the system. Core files may be world-readable and can contain sensitive information (like SSH or PGP keys) or chunks of sensitive files (like `/etc/shadow`). On the other hand, core files might be useful to your developers—you may well end up having different rules for core files on development machines as opposed to production.

Linux /etc/sysctl.conf

```
# General parameters
net.ipv4.ip_forward = 0
net.ipv4.tcp_max_syn_backlog = 4096

# Interface-specific parameters
# <*NOTE*> Must also set net.ipv4.conf.default.*
net.ipv4.conf.all.log_martians = 1
net.ipv4.conf.all.rp_filter = 1
net.ipv4.conf.all.accept_source_route = 0
net.ipv4.conf.all.send_redirects = 0
net.ipv4.conf.all.accept_redirects = 0
net.ipv4.conf.all.secure_redirects = 0
```


On Linux, kernel parameters are normally set in `/etc/sysctl.conf` (they can also be set by accessing the values under `/proc/sys` as we'll see in an upcoming exercise). At boot time, the settings in `sysctl.conf` are automatically applied using the command "`/sbin/sysctl -p /etc/sysctl.conf`".

This slide shows the common security-related network parameters you'll want to set on your Linux systems. In general, there are two classes of these network parameters: General parameters like `ip_forward` and `tcp_max_syn_backlog`, which apply globally to the system, and interface-specific parameters like `rp_filter`, which must be applied to individual network interfaces. The good news is that you can set the "`all`" version of these parameters to apply interface-specific settings to all network interfaces on the system. The bad news is that this only applies these settings to the currently active network interfaces on the system—if network interfaces are enabled later in the boot process, then your settings won't be applied. For this reason, you also want to set the "`default`" version of these parameters (which aren't shown on the slide because they didn't fit).

What do all these parameters mean? Here's the quick rundown:

- *IP Forwarding* is the property of most TCP/IP stacks which causes multi-homed machines to act as routers unless specifically told to do otherwise. In other words, if you have a web server that has one interface pointed at the internet and another connected to some unrouted network where your databases and app servers live, attackers can use your web servers as a gateway to attack the unrouted network unless you specifically disable IP Forwarding (as we're doing here). Note that while the `sysctl.conf` file on every Linux system I've ever seen has IP Forwarding disabled by default, Solaris, and other older proprietary Unix systems typically ship with IP Forwarding enabled by default.
- The `tcp_max_syn_backlog` parameter increases the number of pending "half-open" connections that your system can keep track of, making your system more resistant to SYN flood type attacks. In fact, the modern Linux kernels implement a limited SYN flood protection automatically, but tuning this parameter can still help.

- `log_martians` logs packets with "impossible" source/destination addresses. Martian addresses are defined in Section 5.3.7 of RFC 1812, but essentially mean addresses like network 0, network 127 on non-loopback interfaces, etc. There are reports on the internet of `log_martians` flagging incorrectly configured devices that are attempting to use the LAN broadcast address as their unicast source address.
- `rp_filter` turns on "reverse path" filtering (see RFC 1812). When you turn on `rp_filter` your kernel examines the source address of each incoming packet and looks up the interface it would normally route the return packet out on. If the packet was received on a different interface from the one that the return traffic would normally be sent out on, the packet is dropped as a probable spoof attempt. This can help block traffic like stuff coming in on your Ethernet interface sourced from address 127.0.0.1 (which would normally route through the loopback interface).
- Source-routed packets have a special IP option set that force the packets to go through a specific series of network gateways. Source routing is never used on the modern internet except by attackers who are trying to circumvent your firewalls or IDS/IPS systems, so you should always drop source-routed packets.
- ICMP redirects were designed to allow network gateways to tell hosts about better routes to specific destinations, but they're not typically used much in modern network architectures, which tend to have "star" topologies where each host only has a single route to the rest of the network. The bad news is that ICMP redirects are completely unauthenticated, so they can be used by attackers to route your outgoing traffic incorrectly—typically so that the attacker can intercept the traffic and act as a "man in the middle." Setting `send_redirects` to 0 will prevent your systems from emitting ICMP redirects. The `accept_redirects` parameter controls whether your system will accept ICMP redirects from all and sundry, while the `secure_redirects` parameter controls whether you will obey ICMP redirects from gateways that have been statically configured into the kernel routing table by the system admin using the route add command. Since it's trivial for attackers to spoof the IP addresses of your trusted gateways, you're better off completely ignoring all ICMP redirects.

Local Trouble

Resource exhaustion:

- File descriptors
- Processes ("fork bombs")

Core dumps containing sensitive data (also possibly a DoS)

Sometimes, you have a process that doesn't clean up after itself properly and ends up holding open a lot of file descriptors. If this goes on long enough, you might actually run out of "slots" in the kernel to hold open file descriptors—this can often result in a kernel panic. Similarly, if one of your users runs a program that spawns a lot of sub-processes—generally referred to as a *fork bomb* because fork() is the name of the system call used to spawn sub-processes—they can fill up the process table and lock your system.

While both of these scenarios can happen accidentally, they can also be used by attackers as a denial-of-service attack. For example, this little exploit has made its way around the internet several times:

```
# DO NOT RUN THIS SCRIPT
:(){ :|:& };:
```

Here, we have a function named ":" that recursively calls itself twice (plus the initial invocation of ":" immediately following the function def). If you were to cut and paste this into a shell window, your process table would quickly fill up and you would be unable to start new processes, thus locking up your machine. Naïve users fall for this one regularly.

Core dumps are another object that can be found in the local filesystem. As we'll see, these files may contain sensitive data that could become available to normal users on the system. Also, back before disk space was cheap, core dumps were often used to fill up a disk partition (like `/tmp` or a logging partition) as a denial-of-service attack. Since most machines these days have far more disk space than they'll ever need, it's fairly easy to create large enough disk partitions so that this is not an issue. Note that you can only ever have one core file in a directory. If BIND drops a core file in the root directory and then the SSH daemon drops core, the core file from `sshd` will overwrite the core file from the BIND.

Core Files

```
# ps -ef | grep ftp
root  2968    107  1 16:39:53 ?     0:00 in.ftpd
# kill -ABRT 2968
# file /core
/core: ELF 32-bit MSB core file … from 'in.ftpd'
# ls -l /core
-rw-r--r-- 1 root root 409663 Sep  4  2009 /core
# strings /core
     […]
hal:papAq5PwY/QQM:10588::::::
     […]
```


From time to time on a Unix system, you may encounter a file called `core` (a.k.a. a *core dump*). These files are the remains of a system process that has aborted unexpectedly and contain debugging information that can be useful in figuring out why the program died in the first place. In particular, however, core files usually contain a complete image of the memory allocated to the program at the time of the crash. By running the file through the `strings` program, many interesting pieces of information may be gleaned. The problem is that on most Unix architectures, these core files are world-readable, meaning any user could extract this data.

For example, in this slide, we have forced an FTP daemon to dump core. Examining the resulting core file, we see that the daemon has read the contents of the `/etc/shadow` file (in order to authenticate user logins) and these file contents are still in the memory of the program (Unix processes don't tend to clean up memory until they exit). Running `strings` on the core file gives us the contents of `/etc/shadow` which may then be cracked off-line as described in the previous section. Normally, the data in `/etc/shadow` is only readable by the root user.

As you can see on this slide, the owner of a process can send a program the `ABRT` signal using the `kill` program and force the process to dump core. However, since most "interesting" processes are running with root privileges, this mechanism is seldom used by attackers. Programmer errors often generate core files—often during the development and debugging cycle. However, lingering bugs can cause programs to dump core after the application has been deployed. "Unexpected" input (very long input strings, null strings, strings containing control sequences) can sometimes cause programs to dump core. Buffer overflows may also sometimes cause core dumps.

/etc/security/limits.conf

```
# Prevent core dumps globally
*         hard     core          0

# Other per-user limits
*         soft     nproc         256
*         hard     nproc         512
*         soft     nofile        256
*         hard     nofile        1024
```


Linux system resource controls can be enforced by the `limits.conf` file. Rather than being enforced by the kernel, this file is actually the configuration file for a system PAM module (`pam_limits`). Thus, these limits only apply to interactive users who log into the system, and not to system processes started at boot time or via `inetd` or `cron`. Note also that the limits set in `limits.conf` do not apply to the superuser.

Limits can either be "`hard`" or "`soft`." "`soft`" limits can be overridden by users with the `ulimit` command, but only up to the maximum value allowed by the corresponding "`hard`" limit. Here, we're applying the limits to all users ("`*`" in the first column), but you can actually set limits for specific users and/or specific groups ("`@`<*groupname*>") if you want.

Here, we're preventing user processes from creating core files by forcing a hard limit of 0. If you made this a "soft" limit, then the default would be no core files but individual users could override this (though most users don't know enough to do this).

We're also setting limits on the number of processes per user and open files per process to help stop fork bombs as well as limiting the number of open file descriptors per process.

But What About Boot Services?

- Set `DefaultLimit*` params in `/etc/systemd/system.conf`
- Can be overridden in unit files with `Limit*` parameters
- Use `functions` file on RHEL6 and earlier

If `limits.conf` only controls settings for interactive user logins, what can an administrator do about processes started at boot time? `systemd` includes configuration parameters that will apply to services started at boot time. In `/etc/systemd/system.conf`, you can set a variety of `DefaultLimit...` parameters. For example, `DefaultLimitCORE=0` would prevent services from creating core files. `DefaultLimitNOFILE` would control the max number of file descriptors processes could open, and so on. There are corresponding `LimitCORE`, `LimitNOFILE`, etc. parameters that may be set in individual unit files to override the defaults from `system.conf`.

On pre-`systemd` Red Hat systems (RHEL6 and earlier), all of the boot scripts include a file named `/etc/rc.d/init.d/functions`. Since this file is read in by each and every boot script on the machine, any settings put in this file will apply to all processes started at boot time.

So, the administrator can add `ulimit` commands to the `/etc/rc.d/init.d/functions` file to enforce various settings. The default `functions` file that ships with Red Hat already has a "`ulimit -S -c ${DAEMON_COREFILE_LIMIT:-0}`" command embedded in it. Unless the administrator sets `DAEMON_COREFILE_LIMIT` to some non-zero value in `/etc/sysconfig/init` ("`DAEMON_COREFILE_LIMIT=unlimited`" is the typical setting), processes started at boot time will not dump core. Of course, this means that if you ever need to get a core file from a certain daemon, you'll have to modify the /etc/sysconfig/init file and restart the daemon to remove this limit.

Note that since `cron` and `xinetd` are both started via this same boot-time mechanism, limits set in `/etc/init.d/functions` will apply to these daemons as well. Also, the child processes spawned by these daemons will also inherit the limit settings from the parent process.

Linux Does Cores Right

- Linux has sensible defaults:
  - Writes core files with privileges of the process owner
  - Core files are mode 0600 by default
- Some hooks in `/etc/sysctl.conf`:

```
kernel.core_pattern = /var/core/core
kernel.core_uses_pid = 1
```

- However, `/var/core` would have to mode 1777

The Linux kernel actually enforces some reasonable defaults when it comes to writing core files. First, the kernel dumps the core files with the privileges of the process owner. One side effect of this is that if a process running as an unprivileged user tries to dump core in a system directory where that user doesn't have write permissions, then no core file gets dropped (even if you wanted to get a core file for debugging purposes). If a core file is written, the mode of the file is set to 0600 so that only the process owner can read the file. This means that for daemon processes running as root, normal users wouldn't be able to read the core dump.

There are a couple of kernel variables that you can set in `/etc/sysctl.conf` that control how and where core files get written. `kernel.core_pattern` is the default name for core files. This variable is set to "`core`" by default, so core dumps end up in the current working directory for the process, but you can set this to an absolute pathname like `/var/core/core` to force core dumps to a specific directory (or `/dev/null` to disable cores entirely). Of course, if all core dump files have the same name, then you'll have trouble with core files trying to overwrite each other, so `kernel.core_uses_pid` tells the kernel to append the PID number to the core file ("`core.`*<pid>*"). The only danger here is a process that keeps forking and dumping core, thus filling up the filesystem with hundreds of core files. Note that `core_uses_pid = 1` is the default setting for RHEL 5.x and later.

Remember, however, that core files get written with the privileges of the process owner. If you're directing all core files to a specific directory, then that directory needs to be writable by every user on the system (better set the sticky bit on that directory!). It's not clear that forcing core files to a global directory like this is all that useful from a security perspective if everybody has to have access to the directory.

In RHEL6 and later, when `abrtd` is running, the `kernel.core_pattern` parameter gets modified by `abrtd` to hook into the `/usr/libexec/abrt-hook-ccpp` program to capture additional debugging information about the aborting process (normally in `/var/spool/abrt`). You will still need to re-enable core dumps (see previous slide) for this to happen. `abrtd` will also dump a regular core dump in the current working directory of the process. You cannot use `kernel.core_pattern` to change the destination of the normal core file when `abrtd` is running.

Wrap-Up

Whew! We've already covered a ton of material. Time to break for the day, but there's so much more yet to come!

What's on Tap for Tomorrow?

- SSH Tips and Tricks
  - SSH for Automated Tasks
  - Port Forwarding
- AIDE Over SSH
- Logging and Monitoring Tools
- Syslog-NG
- Syslog-NG Over SSH

Looking ahead to tomorrow's class, we'll be putting together some building blocks for helping to secure your network. There's a lot of cool SSH functionality that you can use for automated tasks like cron jobs and other scripts. We'll be leveraging this functionality when we discuss my idea for running AIDE scans via SSH to make them less vulnerable to attackers who've compromised the systems that you're scanning.

We'll also learn the ability to tunnel traffic over secure SSH connections, which we'll use to set up a distributed logging infrastructure with Syslog-NG. We'll also look at the basic logging services that are available on a typical Unix system before we dive into the mysterious realm of Syslog-NG.

That's It!

- Any final questions?
- Please fill out your surveys!

Hal Pomeranz
hal@deer-run.com
http://www.deer-run.com/~hal/
@hal_pomeranz on Twitter

Lab Exercise

- Fun with resource limits
- Get some experience with:
  - `sysctl.conf`
  - `limits.conf`

Exercises are in HTML files in two places: On the course USB media and in the home directory of the SANS user in the lab virtual machine. These two locations contain identical copies of the exercise files—it just depends on whether you prefer to look at them from a web browser inside the lab VM (because you're working in full-screen mode) or from your host operating system.

1. Open your web browser
2. Select "File… Open…" from the browser menu (Ctrl-O is the usual keyboard shortcut)
3. This is Day 2, Exercise 5, so navigate to *.../Exercises/Day_2/index.html* and open this HTML file
4. Click on the hyperlink, which is the title of Exercise 5
5. Follow the instructions in the exercise

SEC506.3 Securing Linux/Unix

SANS

Linux/Unix Hardening (Day 3)

Hal Pomeranz
hal@deer-run.com
http://www.deer-run.com/~hal/
@hal_pomeranz on Twitter

Today's Agenda

- Automated Tasks Via SSH
  - *AIDE Scans Over SSH*
- Unix Logging Overview
- SSH Port Forwarding
- Syslog-NG
  - *Syslog-NG Over SSH*

Back on Day 1, I listed logging and monitoring as one of the top three things you should do to protect your Unix systems. Today's course covers a number of different topics related to this idea. We'll also be picking up some SSH "tips and tricks" along the way that are applicable to a wide variety of situations that may come up in your environment.

First we'll look at the general problem of using SSH with automated tasks while still doing strong authentication. This section will look at public-key-based authentication with SSH, the ssh-agent facility for storing keys, and how this all ties together with automated tasks like cron jobs. We'll even circle back around to some extra Sudo tricks at the end of this section.

That will lead us back to an idea I mentioned yesterday—running your AIDE scans remotely via SSH to help stop interference by attackers who may have compromised the system. This will give us a practical example of the concepts in the *Automated Tasks with SSH* section.

Next, we'll turn our attention to the standard logging channels available in typical Unix-like operating systems. This is sort of a "level setting" exercise to pave the way for later material.

Then we'll take a brief side tour into SSH port forwarding before diving into a look at Syslog-NG. Syslog-NG is an Open Source Syslog replacement that supports Syslog-style logging over TCP, which means it can be tunneled over SSH. It also has a number of other useful options that make it appropriate to use as the Syslog agent on a large, centralized log server for your network.

Of course, centralized logging also implies that you have time synchronization across your infrastructure (those time stamps have got to match up!), so I've included an Appendix at the end of the book on setting up NTP, for those of you who are at sites where this hasn't already been done. I've also included an Appendix on a simple log analysis tool called Logcheck, which also includes pointers to other, more sophisticated log analysis engines.

Automated Tasks Via SSH

Once you start administering more than one Unix system, you're very quickly going to need some way for automated jobs to execute on multiple systems, without requiring an administrator to enter a password. So, this section discusses some different tactics for accomplishing this.

Automated Logins: Why?

- Cron jobs and other automatic tasks
- Run commands on multiple machines in parallel
- Config file synchronization (`rsync`, etc.)
- Disk-to-disk backups

One of the key jobs of a good System Administrator is automating repetitive tasks. But what about those tasks that require you to operate multiple systems over the network? You need to come up with some mechanism to allow these jobs to access remote machines—often with elevated privileges—in a way that doesn't compromise your security policy but also doesn't require the administrator to sit there and enter passwords all day.

For example, as an extra level of security, I want to run AIDE remotely from a central, secure machine so that an attacker who compromises one of the systems we're scanning doesn't get access to the configuration database we're using to check the system's configuration. However, in order to do this, we need to understand how to remotely access the system we're scanning and run the scan with root privileges.

As an alternative to less secure network-based information services like NIS and even LDAP, one thing you can do is maintain your `passwd`, `shadow`, `group`, `hosts` files, etc. on a central server and then push those files out to other systems using a tool like `rsync`. Again, the `rsync` process is going to need root-level access on the machines you're pushing the files out to.

You can also do file transfers in reverse—pulling copies of files on remote systems back to a central repository. In fact, disk space has become so cheap at this point that this is a fast and effective mechanism for dumping entire filesystems and making full disk-to-disk backups of remote machines. `rsync` is particularly good for this because it can be used to only copy the files that have changed on the remote system. For basic info on using `rsync` for backups, see:

http://ubuntuforums.org/showpost.php?p=4980160&postcount=4

Some existing `rsync`-based backup solutions include:

http://www.stearns.org/rsync-backup/
http://rdiff-backup.nongnu.org/
http://www.dirvish.org/

Automated Logins! How?

- Use SSH with public-key authentication
- But what about passphrases?
  - Don't use passphrase (not recommended)
  - Store keys in **`ssh-agent`**
- Can also leverage Sudo to help reduce security exposure...

So, it seems like having some sort of automated login mechanism seems like a good idea. How do we accomplish this? These days, the usual answer is to use public-key-based authentication with SSH. Remember our discussion of `PubkeyAuthentication` and "`PermitRootPassword without-password`" from Day 1? Well, this is where that starts becoming relevant.

The difficulty with using this form of authentication with automated tasks is that you're typically prompted for a passphrase when creating your public/private key pair. This passphrase is used to encrypt the private key and you're forced to enter this passphrase every time you want to use the key during the login process. So, it seems like we're just swapping a password for a passphrase but not getting any closer to our goal of a fully automated login.

You can actually choose to use a "null" (empty) passphrase when creating your key pair. On the positive side, this means you can use the key without having to type anything—meaning you can now do automated logins with that key. The downside is that your key is now *not encrypted* and is just sitting around in the clear in a file on disk. If somebody steals this key file, then they instantly have access to all the systems that trust that key (potentially many, many systems on your network).

An alternative is to use the `ssh-agent` mechanism for storing your keys and giving access to the processes that require them. Yes, there are possible attacks against the keys that are stored in the `ssh-agent` process (more on this later), but at least the keys themselves will be encrypted when "at rest" in the filesystem (and on your backup images, etc.).

At the end of this section, we'll also talk about a useful trick for using Sudo with automated tasks. That way your automated jobs can SSH around the network as an unprivileged user and use sudo to get the limited amount of root privileges they need to accomplish their tasks. So now if an attacker steals your SSH keys, at least they don't have immediate unfettered root access to other systems on your network. Plus, you get the audit trail from sudo.

Public Key Authentication

We'll talk about some of the basics of setting up public-key-based authentication. If you've never done this yourself, you may also find this document helpful:
https://www.digitalocean.com/community/tutorials/ssh-essentials-working-with-ssh-servers-clients-and-keys

User Public Key Based Authentication (1)

- User generates keys with **`ssh-keygen`**
- Public key in **`authorized_keys`** file on remote systems
- Login challenge encrypted with public key
- SSH client decrypts challenge, returns result to server

Key management the subject of much ongoing research...

To start with, let's talk about setting up public key based authentication for normal, interactive user logins.

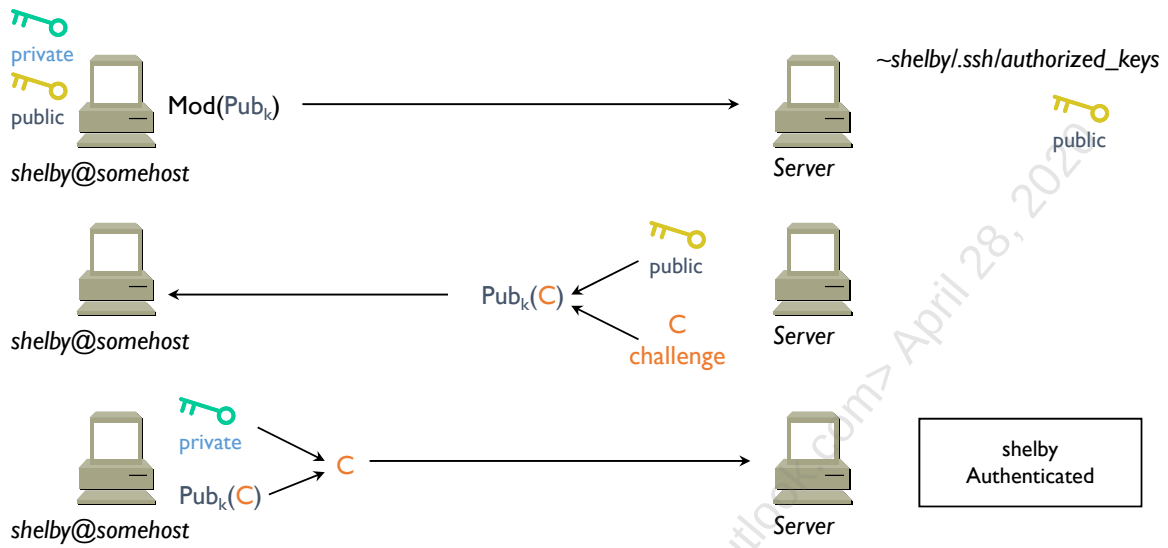
The first step is for a user to generate a public/private key pair using the `ssh-keygen` command. The private key is stored in an encrypted file in the user's home directory (actually in the user's `~/.ssh` directory). The encryption key for this file is a passphrase the `ssh-keygen` program gets the user to enter. A copy of the public key is also placed in the same directory. The user must somehow get this public key into a file called `authorized_keys` on the remote system they wish to log into—either by logging in themselves with a regular password or getting the administrator of the remote host to set the file up for them.

Once the key is set up in `authorized_keys`, the remote server can use this key during future login attempts to encrypt a "challenge"—think of this as a blob of random data. The server sends the encrypted challenge to the client. The SSH client prompts the user for their passphrase, unlocks the private key long enough to decrypt the challenge, and sends the decrypted challenge back to the server to verify the user's identity.

Of course, if somebody steals your private key (aka your identity certificate) and is able to crack it off-line, they can impersonate you to the remote server. As a more secure key storage mechanism, OpenSSH has some support for smartcards already. Check out the configure options `--with-sectok` and `--with-opensc` in the OpenSSH documentation. Another area of research is ways to distribute user public keys (LDAP? NIS+?) because having to manually copy your public key to an authorized_keys file doesn't work really well for thousands of machines in a big network.

One other note is that OpenSSH and the commercial third-party version of SSH use incompatible public key formats. Happily, the version of `ssh-keygen` that comes with OpenSSH can convert back and forth between the two formats (check out the `-e` and `-i` options in the manual page).

User Public Key Based Authentication (2)



This is a simple picture of the process described on the previous slide.

First, user shelby creates a public/private key pair with `ssh-keygen`. She then copies the public key over to the `authorized_keys` file on the server.

Then, when Shelby connects to the remote server, the server encrypts a challenge with Shelby's public key and sends the challenge to Shelby.

Shelby is prompted for her passphrase, which allows her SSH client to unlock her private key and decrypt the challenge. The decrypted challenge is returned to the server and Shelby is authenticated. Hooray!

Licensed To: James Byrd <jeffreybubrow@outlook.com> April 28, 2020

Setting up RSA/DSA Auth

Use "**ssh-keygen -t [rsa|dsa]**" to create your key pair

- Public key is **~/.ssh/id_[rd]sa.pub**
- Don't forget your passphrase!

Get public key into **authorized_keys** file on remote system

- Multiple keys are allowed, comments too!
- Be careful about cut and paste creating extra line breaks—authentication will fail!

As mentioned earlier, generating your public/private key pair is accomplished with the `ssh-keygen` command. You can specify the type of key to generate with "`-t`" and the number of bits in the key with "`-b`". The defaults are "`-t rsa -b 4096`", a 4096-bit RSA key, and this is usually considered sufficient, though you can use longer key lengths if you like.

Here's a sample output from the `ssh-keygen` command where I'm creating a 4096-bit key:

```
$ ssh-keygen -t rsa -b 4096
Generating public/private rsa key pair.
Enter file in which to save the key (/home/hal/.ssh/id_rsa):
Enter passphrase (empty for no passphrase): <not echoed>
Enter same passphrase again: <not echoed>
Your identification has been saved in /home/hal/.ssh/id_rsa.
Your public key has been saved in /home/hal/.ssh/id_rsa.pub.
The key fingerprint is:
14:28:e2:f1:0e:5f:40:58:88:b4:41:91:48:fe:1a:39 hal@foo
```

As you can see, the resulting key files are put into your `~/.ssh` directory.

The `id_rsa` file is the encrypted secret key file. If you enter a passphrase, you will be required to enter that passphrase every time you want to use this key to log in someplace. So, you have the option of just hitting "`<Enter>`" twice when prompted for the passphrase (or run "`ssh-keygen -N '' ...`" option to specify a null passphrase on the command line). This will give you an insecure, *unencrypted* key file, but one that is easier to use for automated logins.

The `id_rsa.pub` file contains the public key that needs to go into the `authorized_keys` file on the remote system. The `authorized_keys` file may contain multiple keys (after all, you might have one

key on your laptop and another on your primary desktop machine, etc.), one key per line. Comments (preceded by '#') and blank lines are also allowed. The keys themselves are very long—typically wrapping over many 80-character lines when you view the file. Be careful if you "cut and paste" the key from your `id_rsa.pub` file into `authorized_keys`, because you might accidentally introduce some new lines into the `authorized_keys` file that will cause you to be unable to authenticate to the remote system!

The Options Field

- Options can be set on keys in the **`authorized_keys`** file
- Comma-separated list at the *beginning* of each line
- Lines are parsed on whitespace:
  - No whitespace after commas!
  - Quoting is *very important!*

What's not immediately obvious is that there's actually an additional field you can set on each key in the `authorized_keys` file. This field allows you to set options that apply to the given key, and these options can limit how the key can be used, which is useful when you're thinking about using these keys for certain specific automated tasks.

The options field is the *first* field on each `authorized_keys` line, but if no options are set (and `ssh-keygen` doesn't set any options by default when creating the key) then the field is null. Multiple options can be set, separated by commas.

The tricky part is that each line in the `authorized_keys` file is parsed on whitespace. You have to be careful not to put any whitespace around the commas that are used to separate options. Also, you must be careful to quote the values of any options you set, just to make sure you don't introduce whitespace. If you introduce unquoted whitespace into the options field, the key will not be parsed properly and again your authentication will fail.

authorized_keys Options (1)

from="<*pattern*>,..."

Restricts access to connections from host names that match specified patterns. Wildcards (*,?) and negation (!) are allowed.

command="*cmd arg...*"

Runs specified command on login, then terminates session. Command line from client is ignored (although **$SSH_ORIGINAL_COMMAND** is set on server side with original command line).

The "from=" option lets you specify one or more host name or IP address patterns. You can use wildcards like "*.foo.com" to match all hosts in a particular domain or "192.168.*" to match by IP address. You can also use statements like "*.foo.com,!bar.foo.com" to specify exceptions. If "from=" is set, then authentication on this key will only be valid if the connection originates from a hostname matching the given patterns. While hostnames and IP addresses can be spoofed, this option adds a little extra bit of security and makes the job of an attacker who steals your key that much harder.

"Forced commands" are set using the "command=" option. The argument to this option is a complete command line that will be executed whenever somebody logs in using the given key. Once the command is completed, the session terminates. Any commands from the user's SSH client are completely ignored. This option is useful for granting privileged access to run specific commands, particularly by automated scripts running out of cron, etc.

The only problem is that the command has to be completely specified. Take, for example, the problem of trying to run a dump via a forced command; command="dump 0f - /usr". Yes, you could dump the /usr partition this way, but you'd have to have separate keys for all of the other partitions you wanted to dump because there's no way to easily specify the partition to be backed up—it has to be hard-coded into the "command=" option. One approach is to just have a single dump script that dumps all the system's partitions, and then invoke that script via the "command=" option rather than calling dump directly.

It's worth noting that the command the user types on their SSH client is put into an environment variable called $SSH_ORIGINAL_COMMAND, which is available to the process running on the server. Theoretically, the user could specify a partition name in the command line on the client side, and a script run via the "command=" option could parse the $SSH_ORIGINAL_COMMAND variable, get the partition name out of it, and dump the appropriate partition. Obviously, security validation on the part of the script parsing the $SSH_ORIGINAL_COMMAND variable is *critical*.

authorized_keys Options (2)

no-pty

Makes interactive shell access impossible because no pty resources will be allocated to connection

no-port-forwarding

permitopen="*host*:*port*"

Disable TCP port forwarding when this key is used, or restrict forwarding to specific destinations. Use multiple ***permitopen*** *options if desired.*

no-X11-forwarding

no-agent-forwarding

Optionally disable other functionality

Other options allow you to restrict functionality for sessions authenticated by a given key. The "no-pty" option is powerful because it prevents malicious users from getting interactive shell access—all they can do is SSH in and run a command, and if they try to shell escape from that command, they can't get an interactive shell because there's no pseudo-tty associated with the session to handle the interactive shell.

Similarly, X11 forwarding, TCP forwarding, and even port forwarding can be prevented if the login is done via a given key (we'll talk about agent forwarding shortly). As far as port forwarding goes, OpenSSH also supports the "permitopen=" option that allows port forwarding, but only to the destination host and port specified by the option. For example, you could let users SSH into some restricted network, but only allow them to set up port forwarding to port 80 on a particular web server machine on that restricted net. You can have multiple "permitopen=" options on a single key.

Here's an example of an authorized_keys line with all options set. Most of the key is not shown here, just to make things easier to read (normally, you'd see an additional few hundred characters of the ASCII encoded DSA key instead of the "…"):

```
from="backups.mydomain.com",command="/usr/local/backup",no-pty,no-port-
forwarding,no-X11-forwarding,no-agent-forwarding ssh-dss
AAAAB3NzaC1kc3MAAACBAPgP… root@backup
```

ssh-agent

`ssh-agent` was really created as a convenience feature for users and sites that make heavy use of public-key-based authentication. Obviously, the usual tension between "convenience" and security applies here—there are some potential security risks to using `ssh-agent`. We can also make use of `ssh-agent` as an alternative to having unencrypted keys on disk for automated jobs.

Security vs. Ease of Use

- Key-based authentication requires retyping passphrase
- **`ssh-agent`** is a program that stores keys in memory
- **`ssh-agent`** does decryptions—no passphrase prompts!
- Downside is potential attack on keys sitting in memory

If you use public-key-based authentication a lot for interactive logins, it can get frustrating to keep having to type your passphrase repeatedly to unlock your secret key. You might ask if there isn't some way to just unlock your certificate once at the beginning of your session and not have to keep retyping it (single sign-on). Well, `ssh-agent` is exactly the utility that does this.

Basically, `ssh-agent` is a key storage program that you run on your local desktop machine. You load your key(s) into the memory of the `ssh-agent` process where they're stored, essentially in the clear. All of the other SSH client programs know how to find and communicate with the `ssh-agent` process via an internal socket on your machine. So rather than prompting you for your passphrase, the client just passes login challenge from the remote server to the `ssh-agent` process, which decrypts it and passes it back to your client. The client can then log you in automatically.

The downside here is that anybody who can break into your machine and get access to your user ID can also talk to the `ssh-agent` process and use it to log in as you on the remote system. Or, if they break root on the system, they can just dump the memory of the `ssh-agent` process to steal your key directly. Remember, the trade-off here is convenience vs. security.

Starting `ssh-agent`

- Normal method is to have **`ssh-agent`** start up your windowing environment
- This allows **`ssh-agent`** process to set magic **`SSH_AUTH_SOCK`** variable
- Then run **`ssh-add`** to store keys in the memory of **`ssh-agent`** process

For `ssh-agent` to work with your SSH clients, the clients need to be able to find and talk to the internal socket used by `ssh-agent`. The mechanism for handling this is the magic `SSH_AUTH_SOCK` variable that is set by the `ssh-agent` process. This variable needs to be set in all of the shells and sub-shells where you want to take advantage of the keys in the memory of `ssh-agent`.

Normally, the way you populate this variable into all of your shells is to start `ssh-agent` *before* you start your normal windowing environment. In fact, the `ssh-agent` process is used to fire up your normal X environment when you log in. This means that all of your X applications are child processes of `ssh-agent`, so if the parent `ssh-agent` process sets an environment variable like `SSH_AUTH_SOCK`, all of your other X apps will inherit this setting. Most Linux distros these days automatically start up `ssh-agent` when starting your windowing environment, so you don't have to do any special configuration at all, but on other systems, you may have to do something like "`ssh-agent startx`" to start your windowing environment via `ssh-agent`.

Note that once you get `ssh-agent` running, you still have to load your keys into its memory. The `ssh-add` program is used for this purpose—just run `ssh-add` with no arguments and you'll be prompted for your passphrase. Assuming you enter the passphrase correctly, your key will be loaded into the `ssh-agent` process. Now you should be able to SSH to some remote system and get logged in without ever having to type your passphrase again (assuming you have your `authorized_keys` file set up properly on the remote system).

Note that you can remove keys from the keyring using the "`ssh-add -d`". You can also set a timeout value for a key in the keyring using the "`-t`" option. For example, "`ssh-add -t 7200`" would add the key from your `~/.ssh` directory and then time it out and remove it two hours later.

fanout and fanterm

- Run commands in parallel on multiple systems via SSH:
  - **fanout** – control non-interactive processes
  - **fanterm** – for interactive jobs (like editors!)
- Requires the use of **ssh-agent** so that authentication doesn't need password
- Can do a lot of damage in a very short amount of time!

The first thing that `ssh-agent` enables is the ability to use any one of several tools that allow you to run commands on multiple machines in parallel via SSH. For example, see `shmux`, available from *http://web.taranis.org/shmux/*, and Cluster SSH, which is at *http://sourceforge.net/projects/clusterssh/*

One of our SANS Faculty members, Bill Stearns, has written a tool like this called "`fanout`" (*http://www.stearns.org/fanout/README.html*). "`fanout`" collects the output from each machine and displays it in an easy-to-read format:

```
$ fanout "deer faun" uptime
Starting deer
Starting faun
Fanout executing "uptime"
Start time … 12:21:58 …, End time … 12:22:02 …
==== On deer ====
   12:22pm  up 139 day(s),  2:19,  5 users,  load…
==== On faun ====
   12:14:32 up 67 days, 7 min,  1 user,  load…
Exiting fanout, cleaning up...done.
```

"`fanterm`" goes this idea one better, by automatically popping up an `xterm` on each remote system plus a local "control" terminal. Anything you type in the "control" window happens in the other `xterms`. This means not only can you run commands on all of these machines simultaneously, you can also do truly bizarre things like edit the same file on all of the client systems at the same time using `vi` or `emacs`!

Obviously, you need to be *very, very careful* when using a utility like this, because you can blow up dozens of systems with one badly typed command.

Agent Forwarding

- Normally, **`ssh-agent`** only helps with "one hop" logins
- *Agent forwarding* allows "multi-hop" logins:
  - Enable agent forwarding with "**`ssh -A …`**"
  - **`authorized_keys`** on all systems contains public key from primary desktop
- Creates a true single sign-on solution, but compromised keys are *deadly*

But now suppose you lived an environment where you had to go through several "hops" to get to some remote system. For example, you might have to SSH from your desktop to some other system to SSH out of your home network. From there, you might have to hop to an inbound SSH server at a co-lo. And from there you might have to hop to the database machine you were actually trying to reach in the first place. At each one of these "hops" you may have to set up a public-private key pair for authentication, and then copy the public key to the `authorized_keys` file on the "next hop" server. This gets to be a hassle—not only to set up but also as far as typing all of those passphrases.

Agent forwarding allows you to use the keys in the `ssh-agent` process *on your primary desktop machine* automatically across multiple "hops". The first step is to set up `ssh-agent` on your primary desktop as previously described and to load your key with `ssh-add`. Now make sure that the public half of that key is installed in the `authorized_keys` file on *all* of the machines you're logging into, no matter how many "hops" away they are. To activate agent forwarding, be sure to specify the "`-A`" flag on all of your SSH client command lines.

When you SSH into a remote machine with `-A`, the remote server sets the magic `SSH_AUTH_SOCK` variable. So, any SSH client program you run on that remote machine ends up talking to your SSH server process on that machine, which relays the authentication request down to your encrypted SSH login session back to the `ssh-agent` process running on your primary desktop machine. This works through multiple hops. It's true single sign-on for SSH.

The only problem is that if somebody manages to steal your identity certificate, then they potentially have access to *all machines* that have the public half of the key in your `authorized_keys` file. This is a significant security risk (convenience vs. security again… in spades).

Lab Exercise

- Fun with public key authentication
- Practice setting up public keys
- Practice using **`ssh-agent`** for personal accounts

Exercises are in HTML files in two places: On the course USB media and in the home directory of the SANS user in the lab virtual machine. These two locations contain identical copies of the exercise files—it just depends on whether you prefer to look at them from a web browser inside the lab VM (because you're working in full-screen mode) or from your host operating system.

1. Open your web browser
2. Select "File… Open…" from the browser menu (Ctrl-O is the usual keyboard shortcut)
3. This is Day 3, Exercise 1, so navigate to *.../Exercises/Day_3/index.html* and open this HTML file
4. Click on the hyperlink, which is the title of Exercise 1
5. Follow the instructions in the exercise

ssh-agent and Scripts

1. Run **ssh-agent** interactively
2. Save shell code output to a file
3. Then "source" this file into all your scripts

- *Setting up **ssh-agent**, adding keys still a manual process*
- *Have to do this on every reboot…*

One of the interesting features of `ssh-agent` is that if you run it interactively from the command line, it outputs the shell code necessary to set the magic `SSH_AUTH_SOCK` variable:

```
$ ssh-agent
SSH_AUTH_SOCK=/tmp/ssh-GvzgWK5920/agent.5920; export SSH_AUTH_SOCK;
SSH_AGENT_PID=5921; export SSH_AGENT_PID;
echo Agent pid 5921;
```

The idea here is that if you just want to run `ssh-agent` within a single window (like when you want to use `ssh-agent` on a remote system you just logged into in an xterm), you can do "`eval $(ssh-agent)`" to easily get this set up.

In our case, however, we can make use of this functionality to set up an `ssh-agent` process that our automated scripts can use. First, we capture the output from the `ssh-agent` process into a file, like so:

```
$ ssh-agent | grep -v echo > ~/.ssh/ssh-agent
```

Notice that I'm filtering out the "`echo`" command from the `ssh-agent` output—this is simply for convenience so that we don't get spurious output every time we use the contents of this file in our scripts.

The next chore is to use `ssh-add` to load some keys into this process. We'll use the "`.`" operator to read in the settings from our `~/.ssh/ssh-agent` file into the current shell so that `ssh-add` will end up talking to the right `ssh-agent` process:

```
$ . ~/.ssh/ssh-agent
$ ssh-add
Enter passphrase for /home/hal/.ssh/id_rsa:
```

Finally, you need to change your scripts so that they also get the appropriate settings from the `~/.ssh/ssh-agent` file. But this is simply a matter of adding "`. ~/.ssh/ssh-agent`" (or whatever pathname is appropriate) near the top of each script.

The good news is that using `ssh-agent` in this way means that you can protect the secret key files with encryption—preventing somebody who steals your disk drives or backup images from immediately accessing the key and using it to compromise other systems on your network. The downside is that you need to remember to run `ssh-agent` and `ssh-add` every time you reboot your system or your cron jobs will start failing. Here's a simple script you can use to do this:

```
#!/bin/sh

export PATH=/bin:/usr/bin

rm -f $HOME/.ssh/ssh-agent
ssh-agent | grep -v echo >$HOME/.ssh/ssh-agent
chmod 600 $HOME/.ssh/ssh-agent
. $HOME/.ssh/ssh-agent
ssh-add
```

One Last Sudo Trick

- Don't want cron jobs to SSH as root?
- Create an unprivileged user and use `NOPASSWD` option:

```
Defaults                !visiblepw
cronuser                ALL = NOPASSWD: /etc/cronjobs/*
```

- Restrict commands to avoid potential privilege escalations

At many sites, they use the techniques we've talked about so far to allow automated tasks to access systems across the network with root privileges. But perhaps you're uncomfortable giving away unfettered root access—especially to a key that's not protected with a passphrase.

sudo has a `NOPASSWD` rule modifier that was clearly designed for these kinds of automated tasks. Instead of creating SSH certificates for the root account, you could set up an unprivileged user (I call this account "cronuser" in the example, but you can use any username you want) and set up a rule in the `sudoers` file similar to the one above.

However, remember our previous discussion of the `!visiblepw` option? This option would normally prevent a process from doing "`ssh` *`somehost`* `sudo` *`somecommand`*" because the user's password would be exposed. However, since the jobs run as cronuser will never require a password, `!visiblepw` will allow the cronuser to "`ssh` *`somehost`* `sudo` *`somecommand`*", which is often what you want to do in cron jobs.

Note that the sample rule on the slide only allows the cronuser to run scripts and programs from a single directory maintained by the administrator. Not only can you put your own locally developed scripts into this directory, but you can also make links (both symlinks and hard links work) to OS programs from this directory. But you have to be careful—if you allow the cronuser to run OS commands like `mv`, `cp`, or `ln`, then a malicious user with access to the cronuser certificate could add other programs to the `/etc/cronjobs` directory and gain escalated privileges.

If you allowed the cronuser to run "`ALL`" programs via sudo, then things wouldn't be much different than simply allowing direct root access—although you would at least be getting the regular sudo audit trail.

Lab Exercise

- Automation with public key authentication
- Options for automating tasks, including **`ssh-agent`**

Exercises are in HTML files in two places: On the course USB media and in the home directory of the SANS user in the lab virtual machine. These two locations contain identical copies of the exercise files—it just depends on whether you prefer to look at them from a web browser inside the lab VM (because you're working in full-screen mode) or from your host operating system.

1. Open your web browser
2. Select "File… Open…" from the browser menu (Ctrl-O is the usual keyboard shortcut)
3. This is Day 3, Exercise 2, so navigate to *.../Exercises/Day_3/index.html* and open this HTML file
4. Click on the hyperlink, which is the title of Exercise 2
5. Follow the instructions in the exercise

AIDE via SSH

The most difficult part of deploying AIDE or any file integrity assessment tool is figuring out how to protect the tool binary itself and your integrity database. This section covers one solution that I've used with some success. Besides, the solution depends on SSH, and it gives us a chance to reinforce some of the concepts from the previous SSH section.

All of the scripts we'll be discussing here are available here:

http://www.deer-run.com/~hal/aide/

Basic Overview

Idea: AIDE checks run from central server via SSH:

- Use `scp` to move AIDE binary/database to remote system
- SSH to remote host, run scan, save output
- Pull updated DB back to server
- Remove all data from remote host

Undetectable except when the scan is actually running!

A clever attacker who breaks into your systems is going to go looking for the directory where your integrity tool and database reside. And they're going to go looking for the cron job that runs the integrity checks on the system. But what if none of this was actually on the machine?

With my solution, the AIDE binary and databases for all systems live on a central, highly secure system. This system has remote root access to the other machines on the network using SSH with public-key-based authentication. This way, a `cron` job on the central server can: (1) copy the AIDE binary and database to the system you want to check, (2) run the scan and capture the output, (3) pull the updated database for the machine back to the central server (if necessary), and (4) remove all traces of AIDE from the host you just scanned.

Unless the attacker happens to be on the machine at the time you're running the scan, there are no traces of AIDE actually being run on the machine. I've also had some thoughts on how to disguise the scan while it's running—more on this later.

SSH Configuration

- Problem statement:
  - Need to reach remote machine as root
  - Want strong authentication
- Configuration:
  - Use **`ssh-agent`** to store keys on central server
  - Install public key on remote system(s)
  - Set "**`without-password`**" option on remote system(s)
- Test setup before proceeding!

We're going to be SSHing from our secure server and running with root privileges on the remote system. Strong authentication is a must. In this case, we'd like to use public-key-based authentication. However, we'd also like to be able to trigger scans to run automatically via `cron`—this means there will be nobody there to actually enter the passphrase to unlock the private key on the server.

So, we'll need to use the tricks we discussed in the previous section to set up automated root logins using RSA keys. You can either create your key with a null passphrase (not recommended) or follow my instructions from earlier to set up an `ssh-agent` process to store your key(s).

On the client side, be sure that the SSH server has "`PermitRootLogin`" set to "`without-password`" so that public-key logins will be allowed, but logins with reusable passwords will be denied. Of course, you also need to put the public half of the key you generated on the central server into the `authorized_keys` file on the machine(s) you plan to scan.

Once you've got all this set up, it's a good idea to test the configuration. You should be able to become root on the central server, set up your `SSH_AUTH_SOCK` as appropriate, and then SSH as root directly into one of the client(s) without providing a password or passphrase. If you've got this much working, you can move on with the AIDE setup.

Client Configuration

- Pick AIDE install directory on client:
  - Will hold **`aide`** binary, config file, and DB
  - Don't scan install directory with AIDE!
- Install AIDE on client, create initial DB
- Create tar archive of install directory
- Copy tarball to central server as <*hostname*>**`.tgz`**
- Scrub all AIDE files off of client

For simplicity's sake, my solution just bundles up the AIDE binary plus the config file and AIDE database for a given machine into a `gzip`-ed tar file. The tarballs for the hosts being scanned just get dumped into a holding directory on the central server. Each tarball is identified by the hostname of the machine being scanned: `<hostname>.tgz`.

I find that the easiest way to create this tarball is on the client system itself. Pick a directory where you want the AIDE scans to run from. Since our automated process is going to be creating and removing this directory every time a scan runs, it's a good idea to locate this directory in a part of your filesystem not covered by the AIDE scan. Otherwise, you'll get false-positives from AIDE because your installation directory is constantly "changing."

Having decided on the installation directory, create your AIDE configuration file. Remember that the values of "`database-file`" and "`database_out-file`" in the AIDE config should match the installation directory you've chosen. Once you've got a configuration file, run "`aide --init`" to build the database and move the new database into the normal location as specified by "`database-file`".

At this point, you should be ready to go. Make a tarball of the installation directory you've just set up (tar the directory holding the AIDE files, not just the files themselves). Copy this tarball to `<hostname>.tgz` on the central server. Then delete all AIDE files from the client machine, including the tarball.

Three Tools and a Config File

Primary script, **`check`**, runs a remote scan on a single host

Dealing with tarballs is cumbersome, so:

- **`update`**: Script updates AIDE DB in tarball
- **`modify-config`**: For making changes to AIDE config file

Common path/filenames into variables in separate file, **`DECL`**

When I started out down this road, there was just a single script called `check` that was responsible for copying the tarball to the remote system, unpacking it, running the scan, grabbing the updated database, and then removing all files from the remote system. But, of course, I ended up needing a script to `update` the tarball for a machine with the new database whenever changes were made to the remote system. I also ended up wanting a script to make changes to the AIDE configuration file for a given machine without having to manually unpack the tarball (`modify-config`). So, I wrote those scripts as well.

The scripts share a lot of common path and filename settings. Rather than having to make updates in all three scripts, I pulled the common settings out into a separate file that all three shell scripts could just read in using the "." operator.

The DECL File

```
# Paths and filenames on remote system
#
REM_DIR=/var/spool          # parent inst dir
REM_ARCH=aide.tgz           # remote tarball
REM_IDIR=aide               # AIDE dir
REM_CMD=aide                # aide binary
REM_CONF=aide.conf          # aide config
REM_ADB=aide.db             # aide DB
REM_NEWDB=aide.db.new       # --update DB
```


Here's a sample of that declarations file:

- `$REM_DIR` is the directory on the remote system where the tarball should be unpacked (i.e., the parent directory of the AIDE installation directory)
- `$REM_ARCH` is the name that the archive file (the tarball) should have on the remote system
- `$REM_IDIR` is the directory created by unpacking the tarball (the directory containing the AIDE binary, config file, and database)
- `$REM_CMD` is the name given to the AIDE binary in the tarball
- `$REM_CONF` is the name of the AIDE config file in the tarball
- `$REM_ADB` is the name of the AIDE DB in the tarball (this name should match the value of "`database-file`" in the AIDE config file, `$REM_CONF`)
- `$REM_NEWDB` is the name of the new database that will be produced with the "`aide --update`" option (this name should match the value of "`database_out-file`" in the AIDE config file, `$REM_CONF`)

While all of this configuration may seem a little redundant, it turns out that having all these variables in one place makes it easy for us to obfuscate our AIDE scans with a little tinkering …

The Stealth Version

```
# Paths and filenames on remote system
# (obscure any references to AIDE)
REM_DIR=/var/spool          # parent inst dir
REM_ARCH=conf.tgz           # remote tarball
REM_IDIR=config             # AIDE dir
REM_CMD=configure           # aide binary
REM_CONF=system.conf        # aide config
REM_ADB=conf.db             # aide DB
REM_NEWDB=conf.db.new       # --update DB
```


If an attacker happens to be on the system when our scan is running, then they may be suspicious of a directory like `/var/spool/aide` containing an AIDE binary, configuration file, and database all named `aide*`. Also, they'll see a process running like "`./aide --config=./aide.conf --update`", which is pretty much a dead giveaway.

However, what if you changed all the filenames to something innocuous looking as we've done on the slide above? Is an attacker going to recognize a process listing of "`./configure --config=./system.conf --update`" as a file integrity scan? Probably not. So, by exporting all of these variables, it makes it easier for the security admin to mask their scans on the remote system.

The next page shows the actual code for the `check` script itself. The command-line syntax is "`check <hostname> [<port> [<tarcmd>]]`". Here *<hostname>* is the name of the machine to scan (and also the name of the tarball stored on the central server). You can specify *<port>* if your SSH daemon runs on a non-standard port, but it's 22 by default. The `check` script expects to find GNU tar on the system as `/bin/tar`. If this is not the case, then you'll need to specify the SSH port followed by the correct path to GNU tar on the remote machine. For example:

```
check foo.mydomain.com 22 /usr/local/bin/gtar
```

```
#!/bin/ksh
# Copyright (C) Deer Run Associates, 2004. All rights reserved.
# Permission to distribute freely as long as Copyright preserved.
# No warranty expressed or implied. Use at your own risk.

PATH=/usr/bin:/usr/local/bin
export PATH

# Import ssh-agent settings
. /root/.ssh/ssh-agent

# Usage: check hostname [port [tarcmd]]
HOST=$1                         # host to check
PORT=${2:-22}                   # can specify alternate SSH port
REMTARCMD=${3:-/bin/tar}        # path to GNU tar on remote machine

# Local pathname definitions
ROOTDIR=/local/aide             # where tarballs live locally
TEMPFILE=$ROOTDIR/.run$$        # temp file to use for output

# Pick up other settings
. $ROOTDIR/DECL

if [ ! -f $ROOTDIR/$HOST.tgz ]; then
        echo "Tarball $ROOTDIR/$HOST.tgz does not exist!"
        exit 255
fi

cp /dev/null $TEMPFILE

scp -P $PORT $ROOTDIR/$HOST.tgz \
       ${HOST}:$REM_DIR/$REM_ARCH >>$TEMPFILE 2>&1
ssh -p $PORT $HOST "(cd $REM_DIR; \
                     $REMTARCMD zxfp $REM_ARCH; \
                     cd $REM_IDIR; \
                     ./$REM_CMD --config=./$REM_CONF --update; \
                     $REMTARCMD zcf $REM_ARCH $REM_NEWDB)" \
                     >>$TEMPFILE 2>&1
scp -P $PORT ${HOST}:$REM_DIR/$REM_IDIR/$REM_ARCH \
             $ROOTDIR/$HOST.tgz-update >>$TEMPFILE 2>&1
ssh -p $PORT $HOST \
        /bin/rm -rf $REM_DIR/$REM_IDIR $REM_DIR/$REM_ARCH \
        >>$TEMPFILE 2>&1

if [ ! "`grep '### All files match AIDE database.' $TEMPFILE`" ]
then
        awk "!/^Authorized/ && !/^$HOST.tgz:/ { print }" $TEMPFILE
fi
rm $TEMPFILE
```

Future Work

- Make existing scripts more robust and foolproof (root **`rm`** always worrisome)
- Include debugging output option and testing option (ala "**`make -n`**")
- Create meta-daemon for running many tests in parallel
- Better admin/management interface

My scripts in their current form are "good enough" for a small network of machines. For a large, very heterogeneous network, it would probably be a good idea to keep the AIDE binaries, config files, and database files separate, just to make updates simpler all around. For a large network, you'll also probably want some sort of scheduling daemon to handle actually running the scans on the individual machines, and collecting/summarizing the output.

The scripts themselves could also be more robust and error-proof. I actually destroyed a system during my testing because the `rm` command at the end of the check script went awry due to a misconfiguration in the `DECL` file. Having a "show me the commands you would run, but don't actually run them" check would be really helpful for debugging (as would an option for producing more verbose debugging output). Oh well, somewhere in all my "copious free time" I'm sure I'll continue to tinker…

Unix Logging Overview

This section covers configuring Syslog and also discusses other available logging options such as the system accounting facility and process accounting.

The idea is to capture as many different logging sources as possible. An attacker may not adequately cover their tracks if you are logging unexpected information or logging information to an unexpected location. Remember that in *The Cuckoo's Egg*, Cliff Stoll noticed an intruder due to a $0.75 discrepancy in an accounting log.

Classic syslogd

- Different choices for message destination

```
authpriv.info    /var/log/secure    # to logfile
authpriv.info    @loghost           # to remote host
```

- Whitespace MUST be tabs
- Logs severity level info and higher
- OK to log the same messages to multiple destinations

The Syslog priorities, in descending order of importance, are: *emerg, alert, crit, err, warning, notice, info, debug.* Syslog generally will log all messages of the priority you specify ***plus*** all messages of higher priority levels. However, some newer Syslog agents may allow you to specify facility/priority pairs like "`authpriv.=info`" which mean "log `authpriv.info` messages only" (though this is not standard behavior for traditional Unix Syslog implementations).

Log messages can be sent to a local file or another host (note that you can use host names or IP addresses). It is perfectly OK to send the same logging information to multiple destinations. In fact, it is considered good practice to have a copy of your logs on another system to help thwart attacks that tamper with your local log files. Also collecting all of your system logs on a central log server makes analyzing logs more convenient.

Note also that it is vitally important to use **tabs** in `/etc/syslog.conf` or the file will not be parsed properly. Again, newer syslog agents may allow you to use other whitespace, but traditional Syslog insists on tab characters.

Caveats

- Manual log file creation?
- Don't forget about log rotation!

The standard Syslog daemon on traditional Unix systems will not create files on its own. So, if you're going to add a new log file in your `syslog.conf` file, you must create the initial log file yourself. Syslog writes to these files as `root`, so the files do not need to be world-writeable and don't even have to be world-readable if they contain sensitive information (by definition, log data is almost always sensitive). Note that the Syslog daemon that ships with Linux actually will create new log files on the fly, so you don't need to worry about this issue there.

However, one of the problems with starting a new log file is that the administrator must ensure that the log file is "rotated" (moved out of the way and a new log file started) so that it doesn't grow without bound and consume the entire filesystem. Red Hat releases have shipped with a generic log rotation program, `logrotate`, for quite a while but older, proprietary Unix systems may not (e.g. Solaris prior to Solaris 9).

There are a number of "free" log-rotating scripts available on the internet—a Google search for "rotate unix logs" comes back with millions of hits.

If you end up rolling your own log rotation script, you should know that there is a "correct" way to rotate a log file. You're supposed to rename the old log file with the "`mv`" command so that `syslogd` can continue writing to the file (assuming you're moving files within the same partition, `mv` just changes the filename but not the inode number on the file), then create a new file with the canonical log filename, and then `HUP` or restart `syslogd` to get it to close the old file and open the new file you just created. In this way, no logging information is lost.

Gathering System Accounting

System accounting gathers data on system resource usage:

- CPU and memory utilization, paging
- Disk and file I/O, TTY activity
- System calls, semaphore activity, ...

Data archived to `/var/log/sa`

Nightly reports overwritten monthly, raw data deleted weekly

When system accounting is enabled on a Linux system, the machine runs a `cron` job every 10 minutes to capture system performance data. This data is very similar to the data reported by `vmstat`—CPU and memory usage, paging and swapping, disk I/O, etc. The data is logged into files in `/var/log/sa`. Other Unix variants may log to other locations and/or use a different frequency for data collection. For example, Solaris logs data every 20 minutes into `/var/adm/sa`.

Note that the filenames are typically `sa`*dd*, where *dd* is the day of the month—this means that the data will get overwritten on a monthly cycle. Since it's useful to keep this data for longer than a month, you may want to arrange a `cron` job that copies this data to a different directory before it gets overwritten. In fact, the raw accounting data gets summarized into nightly reports and then deleted after a week, so it's probably best to back up data on a weekly (or more frequent) basis.

System accounting data only amounts to a couple of megabytes *per week,* so you can keep lots of data for a long time without using too much disk space. Enabling system accounting under Linux is trivial—just make sure the `sysstat` package is installed.

Why Is This Useful?

- Metrics (impressing management)
- Performance tuning
- Capacity planning, justifying hardware
- Detecting intruders due to unexpected system utilization

System-accounting data can be used to produce metrics, charts, and graphs that can stupefy the average manager into complete insensibility. Trend analysis on this data can be great for justifying new hardware.

From a security perspective, however, the data represents a baseline picture of system behavior. If the system deviates from this baseline, then maybe somebody is using the system in a way that they shouldn't (again, remember Cliff Stoll in *The Cuckoo's Egg*).

For example, suppose you receive daily graphs of your server's CPU usage. From 8 a.m. to 7 p.m. every day, the CPU runs at about 60% and then drops to nothing outside of business hours except for a brief spike during backups. One morning, you come in and look at the graph and you see *two* CPU spikes last night. Isn't *that* interesting…was somebody running a password cracking tool on the system? Perhaps a denial-of-service attack as part of an IP spoofing attack? At least you'll know that *something* strange was happening and can go investigate.

How to Get at the Data

- Nightly reports: `/var/log/sa/sar*`
- Query interactively with `sar`
- Visual SAR
- "Roll your own" with Perl/GNUplot

One of the system accounting `cron` jobs also generates a daily activity report in the `/var/log/sa` directory. The report files are called `sar`*dd*, where again *dd* is the day of the month.

The `sar` command will go against the archived data files in `/var/log/sa` (though you can specify a different filename with the `-f` flag) and produce tabulated reports on the standard output. Note that the system accounting cron jobs have to have been collecting data for a while before you can get anything useful out of the `sar` command.

Visual SAR (*http://visualsar.sourceforge.net/*) is an Open Source Java application that lets you visualize System Accounting data. You can also use the raw data yourself—combined with a tool like GNUplot, you can create lots of pretty charts and graphs for management consumption.

Process Accounting

- Kernel logs data about processes running on the system
- Can be selectively enabled and disabled:

```
# accton /var/log/pacct
# cd /var/log
# ls -l pacct
-rw-r--r--   1 root    root     64 Dec  6 10:40 pacct
# lastcomm -f pacct
lastcomm     root  pts/2   0.00 secs Sat Dec  6 10:40
ls           root  pts/2   0.00 secs Sat Dec  6 10:40
accton   S   root  pts/2   0.00 secs Sat Dec  6 10:40
# accton
```


Process accounting is a mechanism originally developed to allow large data centers to do charge-backs based on system resource usage. From a security perspective, however, having a log of all of the commands run by all of the users on the system is an obvious win (but see warnings on next slide). Note, however, that you just get the command names and none of the command arguments—i.e., just "`vi`" and not "`vi /etc/shadow`".

To enable process accounting, you first have to have the right software installed on your system. On Linux, make sure the `psacct` package is installed.

Process accounting can be turned on and off at will using the `accton` command. To enable process accounting, simply specify a file where the logging should take place (`/var/log/pacct` is standard for Linux). `accton` with no arguments turns off process accounting. Once process accounting is enabled, the kernel logs 64 bytes of data for each process that runs to completion on the system.

The accounting logs are in a binary format that can be dumped with the `lastcomm` command on Linux (sometimes or the `acctcom` command on older proprietary Unix systems). The `lastcomm` command gives a simple reverse-chronological listing of command information. Depending on the flavor of Unix you're using, other commands for analyzing the accounting data (and generating charge-back reports, etc.) may be available. See the `accton` manual page for more information.

Warnings

- Process accounting can cause a 10-20% percent performance degradation
- 40-64 bytes of data logged per process:
  A lot of data on some hosts!
- Data is logged when process finishes:
  Some processes may not be logged

64 bytes of data doesn't seem like a lot until you think about running process accounting on a web server or other server that does a million or more transactions per day. Also, note that simply logging the data can cause a significant performance degradation. In particular, the disk that is the logging device becomes a "hot-spot" in your filesystem—you may have to move to a quicker filesystem type or even an SSD.

Also, note that process logging only happens when the process terminates, so some processes may never get logged (daemons that only terminate when the system reboots, and packet sniffers that attackers are running on your system).

Linux auditd

For a significantly more powerful and granular type of logging, there's the `auditd` system built into the Linux kernel.

Linux `auditd`

Kernel-based activity monitor

Can track a wide variety of activity:

- Program execution
- File access
- System calls
- Keystrokes

The Linux `auditd` system hooks deeply into the kernel, giving it highly granular access to events happening on the system. You can track when files are accessed—whether that's executing a program, or just opening a file for reading or writing. More generally, `auditd` can track any system call (see `/usr/include/asm/unistd_64.h` for a list of system call names), including matching specific system call arguments and error codes. There's even a hook for capturing user keystrokes, though whether this is a feature or a danger is in the eye of the beholder.

RedHat has good documentation on `auditd` in their RHEL Security Guide:

https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Security_Guide/

This is also this nice high-level intro:

http://security.blogoverflow.com/2013/01/a-brief-introduction-to-auditd/

auditd.conf

Where and how logs are written

`log_file`, `log_format`, `flush`

Log rotation

`max_log_file*`, `num_logs`

When space fills up

`*space_left*`, `disk_full_action`

Remote logging

`dispatcher`, `tcp_listen_port`

The `/etc/audit/auditd.conf` file configures general parameters about how the auditing system operates. For example, `log_file` specifies where to write the audit logs—`/var/log/audit/audit.log` is the default. `log_format` is normally `RAW`, meaning to write the audit data to the `log_file` location. However, you can set this to `NOLOG` if you don't want to write local logs because you are sending them off the system. `flush` and `frequently` work together to control how hard your system tries to flush audit data to disk in case of crashes.

Happily, the audit system has its own internal log rotation mechanism. Set `max_log_file` to be the max size of the `audit.log` in megabytes. `max_log_file_action` controls what happens when this size is reached. Typically, the action is either `keep_logs` or `rotate`. Both of these options rotate the logs, but `rotate` will only keep up to `num_logs` log files around while `keep_logs` will hold onto logs even as your disk fills up.

The `auditd` system can send warnings and take actions as your disk starts to fill. There's actually two levels of warnings: `space_left`/`space_left_action` and `admin_space_left`/`admin_space_left_action`. The `space_left` is the threshold of disk space remaining (in magabytes) before `space_left_action` is taken. The `admin_space_left*` parameters are the same, except you can specify a more urgent threshold and action. Actions include sending a warning to `syslog`, sending an `email` to the address configured as `action_mail_accot`, `exec` a script, `rotate` the logs and throw away the oldest to get space back, `suspend` auditing, and even `halt` the system or put it into `single`-user mode for maintenance. You can also use `disk_full_action` to specify what happens when the disk is completely full.

The audit system can route logs to external systems and programs via the `audispd` daemon, configured via the `dispatcher` parameter in `auditd.conf`. You have the option of sending audit logs via Syslog, but `audispd` can also forward logs to `auditd` on another system using a native, TCP-based protocol. See the `audispd` and `auditd.conf` manual pages for further information.

audit.rules (I)

```
## Flush all rules
-D

## Increase the buffers to survive stress events.
-b 8192

## … rules go here …

## Require reboot to change policy
-e 2
```


The `/etc/audit/rules.d/audit.rules` file is where you configure what you want `auditd` to monitor. There are several different sample policy files under `/usr/share/doc/audit-*` that you can use for ideas and inspiration. The CIS Benchmarks also have suggestions for what to audit.

The entries in `audit.rules` are literally the command-line options you would give to the `auditctl` command to change settings on the command line. But maintenance and auditability are easier if you keep everything in `audit.rules`.

Before we get to the rules themselves, this slide shows some "housekeeping"-type items that are normally found in `audit.rules`. The "`-D`" line says dump all current rules and load everything from scratch. "`-b`" specifies the amount of internal buffer space the audit system should have if events are coming in too fast to push everything out to disk (the value on the slide is a reasonable value that comes from the DISA STIG).

The "`-e`" flag can be used to enable ("`-e 1`") and disable ("`-e 0`") auditing. However, "`-e 2`" enables auditing and also locks in the current audit policy so that it can only be changed by rebooting the system. Make this the last line of `audit.rules` if you want to set this.

audit.rules (2)

```
## User authentication files/directories
-w /etc/passwd -p wa -k auth-files
-w /etc/shadow -p wa -k auth-files
-w /etc/security/opasswd -p wa -k auth-files

-w /etc/group -p wa -k auth-files
-w /etc/gshadow -p wa -k auth-files

-w /etc/sudoers -p wa -k auth-files
-w /etc/sudoers.d/ -p wa -k auth-files

-w /etc/pam.d/ -p wa -k auth-files
```


You use "`-w`" to create rules for monitoring specific files and directories. If you specify a directory, the audit system automatically watches everything under that directory (e.g., the last two rules on the slide).

"`-p`" specifies what you want to watch for. "`w`" means anybody making changes ("writing") to the file. "`a`" means monitor for any attribute changes, like timestamps, ownership, permissions, etc.

"wa" is the most common combo of flags because it lets you know when things are changing about the file. If you want to know whenever the file is accessed at all, use "`r`". If you want to track when a program is executed, use "`x`" (we'll see an example of this on the next slide).

"`-k`" lets you specify a keyword tag that will be added to all audit.log entries created by the rule. You can use this for searching your `audit.log` files, as we'll see shortly.

The rules in `audit.rules` are "first match and exit"-type rules. So, the first rule from top to bottom that matches a given file will be applied and no further rules will be consulted.

audit.rules (3)

```
## Kernel module insertion/deletion
-w /sbin/insmod -p x -k kernelmod
-w /sbin/rmmod -p x -k kernelmod
-w /sbin/modprobe -p x -k kernelmod

-a always,exit -F arch=b32
     -S init_module,finit_module -k kernelmod
-a always,exit -F arch=b64
     -S init_module,finit_module -k kernelmod

-a always,exit -F arch=b32 -S delete_module -k kernelmod
-a always,exit -F arch=b64 -S delete_module -k kernelmod
```


The first three rules on this slide show monitoring with "`-p x`" to see if a given executable is being run. If you're also worried about the possibility of rootkits replacing these executables, you might change this to "`-p wax`".

Use "`-S`" to monitor for different system calls. Notice that you can specify a comma-separated list of system calls after "`-S`", but do not put spaces after the commas or you will get a syntax error. You are allowed to use multiple "`-S`" options in the line if you like, or just have multiple rules like we're doing here.

"`-a`" specifies that we `always` want to log these system calls at the time of system call `exit`. Instead of `always`, you can use `never` to create an exception rule to suppress a particular set of system call arguments that are uninteresting. Another way to suppress logging is to use `exclude` instead of `exit`.

Note the use of "`-F`" to specify both the 64- and 32-bit versions of each system call. This is just to be on the safe side for 64-bit architectures, which generally still support the 32-bit system calls for backward compatibility purposes.

By the way, though the lines in the middle of the slide are broken into two lines, *they must appear as a single long line in the `audit.rules` file!*

Loading Rules

- Put files in `/etc/audit/rules.d`
- Load rules with `augenrules --load`
- For scripting: `augenrules --check`

You'll find a file on your system called `/etc/audit/audit.rules`, but this file is actually generated from files under `/etc/audit/rules.d`. The default is to only have a single `/etc/audit/rules.d/audit.rules` file, but you can have as many different files in the directory as you want. This may be a good idea if your audit policy becomes complex—just group related rules together into their own file and then you can add and remove groups of related policy rules more quickly and accurately.

The `augenrules` program reads the files from `rules.d` and smashes them together into the `/etc/audit/audit.rules` file. `augenrules` reads the `rules.d` files in the normal alphabetic sort order used by `ls`, so consider using a naming convention with numbers ("`00rules`", "`05morerules`", "`30evenmore`", etc.) so rules are loaded in the proper dependency order. Remember, `audit.rules` use "first match and exit" behavior, so the order is important!

`augenrules --load` creates `/etc/audit/audit.rules` and loads the new rules into the kernel (assuming you don't have "`-e 2`" set). It will alert you to any syntax errors in your files. Note that `augenrules` is smart enough to put only one "`-D`" and one "`-b`" option at the top of the generated `audit.rules` file. If you have these options in multiple `rules.d` files, it simply uses the last one it sees. Similarly, `augenrules` takes the last "`-e`" option specified and puts it at the bottom of the generated `audit.rules` file.

`augenrules --check` will tell you if any of the files in `rules.d` are newer than the audit policy running in the kernel. This is useful in shell scripts. Note that `--check` *does not* perform syntax checking on the rules files like `--load` does. I wish it did.

audit.log Entries

```
type=SYSCALL msg=audit(1433751377.899:1785): arch=c000003e
    syscall=2 success=yes exit=5 a0=7fc0ce1ed0b0 a1=42 a2=120
    a3=fffffff8 items=2 ppid=4395 pid=40636 auid=1000 uid=0 gid=0
    euid=0 suid=0 fsuid=0 egid=0 sgid=0 fsgid=0 tty=pts0 ses=6
    comm="visudo" exe="/usr/sbin/visudo"
    subj=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
    key="auth-files"
type=CWD msg=audit(1433751377.899:1785):  cwd="/etc"
type=PATH msg=audit(1433751377.899:1785): item=0 name="/etc/"
    inode=130307 dev=fd:00 mode=040755 ouid=0 ogid=0 rdev=00:00
    obj=system_u:object_r:etc_t:s0 objtype=PARENT
type=PATH msg=audit(1433751377.899:1785): item=1
    name="/etc/sudoers" inode=159885 dev=fd:00 mode=0100440 ouid=0
    ogid=0 rdev=00:00 obj=system_u:object_r:etc_t:s0 objtype=NORMAL
```


Here are some raw entries from `/var/log/audit/audit.log`. These logs were generated when I used the `visudo` command to edit my `/etc/sudoers` file. Note that I didn't actually make any changes, but still got all of this logging detail.

In the first entry, you can see that I used `visudo` and the executable path was `/usr/sbin/visudo`. You can even see the process ID of the process I was running as root ("`uid=0`"). At the end of the line, you see the `auth-files` key we had in our `audit.rules` entry.

The next line actually shows you the directory I was in when I ran the command. My `CWD` (current working directory) was `/etc`. The last entry on the slide shows the file that was being accessed—`/etc/sudoers` in this case.

How do we know all of these entries are related to the same event? Note the numbers inside the parens—the "`1785`" after the colon is the unique audit ID of this event and all entries related to this event will be tagged with the same number. The number in front of the colon is actually a timestamp in *Unix epoch time* format (number of seconds since Jan 1, 1970). You can translate these times with the `date` command like so:

```
# date -d @1433751377.899
Mon Jun  8 04:16:17 EDT 2015
```

Obviously, this is less than ideal.

ausearch FTW!

Lots of search options

```
ausearch -f /etc/sudoers
ausearch -k auth-files
ausearch -a 1785
ausearch -ua hal
```

Does timestamp conversion for you!

While the `audit.log` files are just text files and you can `grep` around in them to your heart's content, the `ausearch` command makes many common kinds of searching easier. It also translates the epoch time timestamps to human-readable format for you, so what's not to like?

"`-f`" searches for all entries related to a particular file, while "`-k`" searches for entries by the keywords we set in the `audit.rules` file. Here's some sample output—note the human readable timestamp near the top.

```
# ausearch -f /etc/sudoers
----
time->Mon Jun  8 04:16:17 2015
type=PATH msg=audit(1433751377.899:1785): item=1 name="/etc/sudoers"
   inode=159885 dev=fd:00 mode=0100440 ouid=0 ogid=0 rdev=00:00
   obj=system_u:object_r:etc_t:s0 objtype=NORMAL
type=PATH msg=audit(1433751377.899:1785): item=0 name="/etc/" inode=130307
   dev=fd:00 mode=040755 ouid=0 ogid=0 rdev=00:00
   obj=system_u:object_r:etc_t:s0 objtype=PARENT
type=CWD msg=audit(1433751377.899:1785):  cwd="/etc/audit"
type=SYSCALL msg=audit(1433751377.899:1785): arch=c000003e syscall=2
   success=yes exit=5 a0=7fc0ce1ed0b0 a1=42 a2=120 a3=fffffff8 items=2
   ppid=4395 pid=40636 auid=1000 uid=0 gid=0 euid=0 suid=0 fsuid=0 egid=0
   sgid=0 fsgid=0 tty=pts0 ses=6 comm="visudo" exe="/usr/sbin/visudo"
   subj=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023 key="auth-
   files"
```

As you can see, `ausearch` pulls back all entries related to the same audit ID. You can search for a specific audit ID (if you know it) with "`-a`".

"`-ua`" allows you to search for a user by name or UID. In this case, the "`a`" in "`-ua`" means *all*—it will find entries related to that user ID if it's the real UID, effective UID, or login UID on any entry. There are also other options ("`-ui`", "`-ue`", "`-ul`") if you only want to search for UIDs in a specific field.

There are tons of other search criteria you can use. Read the `ausearch` manual page for more details.

Built-in Keylogger!

Add to `/etc/pam.d` configs:

```
session  required  pam_tty_audit.so
  disable=* enable=root,hal
```

Add `log_passwd` option if desired

View logs with `aureport --tty`

There is a PAM module that can be used to log all keystrokes for selected users called `pam_tty_audit.so`. The keylogger data is sent through `auditd` and ends up in your `audit.log` files. To enable this functionality, add the entry like the one on the slide *as a single long line* in `/etc/pam.d/system-auth` *and* `/etc/pam.d/password-auth`. After this, all new interactive sessions for the named users will be logged. Normally, `pam_tty_audit` will not log passwords and other text that is not echoed back to the terminal. But you have the option of adding `log_passwd` to the end of the line to log this data as well.

Notice the use of "`disable=* enable=root,hal`". In this case, keystrokes for both `root` and `hal` will be logged. But the default assumption is that the keylogger will capture keystrokes for any process started by those users (for example, if I open up a text editor window and start typing). The problem is, what if the `root` user restarts the `sshd` process? Keystrokes will now be logged for *any* user who logs in via SSH! "`disable=*`" sets a default policy that no user keystrokes should be logged, and then the "`enable=`" part overrides that for just the specifically named users.

So, guess what? There is a keystroke logger built into your kernel and there's not much you can do about it. I wonder how many sites wouldn't notice if somebody added the above configuration to their PAM config files? How many sites fail to look at their `audit.log` files?

The nicest way to look at the keylogger data is with `aureport --tty`:

```
# aureport --tty

TTY Report
===============================================
# date time event auid term sess comm data
===============================================
1. 06/08/2015 05:25:35 1956 1000 ? 157 bash <^D>
2. 06/08/2015 05:25:37 1959 1000 ? 157 bash "df",<ret>,"cat
/etc/passwd",<ret>,"/bin/su",<ret>,<^D>
…
```

If you look at the raw `audit.log` entries (just "`grep type=TTY`" or "`ausearch -m TTY`"), you find that the data is stored as a hex ASCII representation. Here are the raw entries corresponding to the `aureport` output above:

```
type=TTY msg=audit(1433755535.406:1956): tty pid=44215 uid=0 auid=1000
   ses=157 major=136 minor=2 comm="bash" data=04
type=TTY msg=audit(1433755537.315:1959): tty pid=43539 uid=1000 auid=1000
   ses=157 major=136 minor=2 comm="bash"
   data=64660D636174202F6574632F706173377640D2F62696E2F73750D04
```

The hex notation means you can't `grep` the `audit.log` for specific command or filenames. But you can "`aureport --tty | grep …`" The only problem with that approach is that backspaces are included in the keystroke logs, so you'll see things like this in the aureport output:

```
62. 06/08/2015 09:56:53 2578 1000 ? 174 bash
"aud",<backspace>,"se",<backspace>,<backspace>,"report --tty",<ret>
```

Good luck `grep`-ing for the `aureport` command there!

Lab Exercise

- Getting to know **`auditd`**
- Get some experience configuring **`auditd`**
- Practice searching and reading audit logs, etc.

Exercises are in HTML files in two places: On the course USB media, and in the home directory of the SANS user in the lab virtual machine. These two locations contain identical copies of the exercise files—it just depends on whether you prefer to look at them from a web browser inside the lab VM (because you're working in full-screen mode) or from your host operating system.

1. Open your web browser
2. Select "File… Open…" from the browser menu (Ctrl-O is the usual keyboard shortcut)
3. This is Day 3, Exercise 3, so navigate to *.../Exercises/Day_3/index.html* and open this HTML file
4. Click on the hyperlink, which is the title of Exercise 3
5. Follow the instructions in the exercise

X11 and TCP Forwarding with SSH

This section looks at some additional useful SSH techniques. In particular, we'll be taking advantage of SSH's TCP port forwarding feature when we talk about building a centralized log server with Syslog-NG a little bit later today.

Basic X11 Forwarding

- SSH can transparently tunnel X11 traffic
- Xauthority information automatically handled
- Important as more apps move to "GUI only" admin

Use the `-X` option to enable X11 forwarding on the command line, i.e. "`ssh -X` *hostname* …". The system administrator may explicitly set X11 forwarding as the default for all sessions by putting something like this in the global `ssh_config` client file:

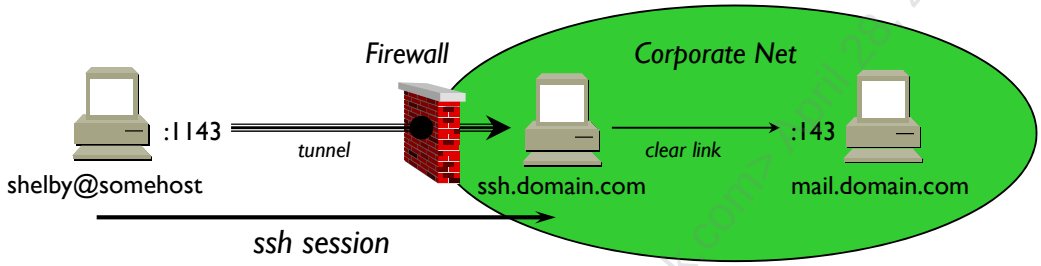
```
Host *
ForwardX11 yes
```

Once X11 forwarding has been enabled, you'll notice that your `DISPLAY` environment variable on the remote system has been set to something like "`localhost:10.0`". Normally this setting would mean that there was an X server listening on port 6010/tcp, but in this case, that port is actually being managed by the remote SSH server that's handling your login session. Because of this `DISPLAY` setting, any X client you run (an `xterm`, `emacs`, or some more complicated GUI) will send all of its display events to this port. The remote SSH server gobbles up this traffic, encapsulates it, and shoves it down the encrypted SSH connection to the SSH client on your local machine. Your client receives the encapsulated traffic, unpacks it, and hands it to the local X Windows server running on your desktop. Voila! Secure X Windows connections from remote systems.

This functionality is becoming increasingly important as more and more applications move to "GUI only" administration. Another advantage to X forwarding is that SSH automatically takes care of `.Xauthority` style authentication issues for your users. Finally, if you're using SSH for all of your X11 traffic, then you can tell your local X server to stop listening on 6000/tcp (via the X server option "`-nolisten tcp`"), reducing the number of open ports on your local desktop system.

TCP Forwarding

- SSH can tunnel any *TCP-based* protocol
- Useful for "quick and dirty" VPNs for specific protocols (IMAP, HTTP, etc.)



ssh –L 1143:mail.domain.com:143 ssh.domain.com

X11 forwarding, however, is really just a special case of the more general port forwarding functionality built into SSH. In general, SSH is capable of forwarding any TCP-based protocol—but not a UDP-based protocol like remote Syslog messages (which is why we'll be talking about Syslog-NG in the next section, because Syslog-NG supports TCP-based logging).

Let's suppose your company had a machine set up behind their firewall that you could SSH into from the outside world. One way you could check your email would be to SSH into this machine, and then SSH from that machine to the host where your mail was stored. Once on the mail server, you could use a command-line-based mail reader to view your email.

But suppose you wanted to use your favorite IMAP-based email client from your laptop? When you SSH into that first system behind your company's firewall, you can use the "`-L 1143:mail.domain.com:143`" option to set up a tunnel via your SSH login session. The first field, "`1143`", says that the local end of the tunnel will be bound to "`localhost:1143`" (so this is what you're going to tell your local email client to use as the "mail server"). The rest of the argument tells the remote SSH server where to send the traffic that comes down the tunnel—in this case to the IMAP port (143/tcp) on the internal machine `mail.domain.com`.

Once you have the tunnel set up and your local mail client reconfigured, everything should just work transparently. The IMAP requests from your mail client to `localhost:1143` should get encapsulated and shoved down the encrypted tunnel. Upon reaching the far end of the tunnel, they should get unpacked and sent *in the clear* across your internal corporate network to the IMAP server on `mail.domain.com`. Now from `mail.domain.com`'s perspective in this example, these requests are originating from the machine `ssh.domain.com`, so all return traffic goes back to this system, where it is encapsulated and sent back to your SSH client, which in turn hands the info off to your waiting mail client. It all just "works."

Canonical HTTP Problem

- It's easy to set up an SSH tunnel to a single web server
- But what if you have to follow a link to some other server?
- Multiple port forwarding entries a hassle and not seamless

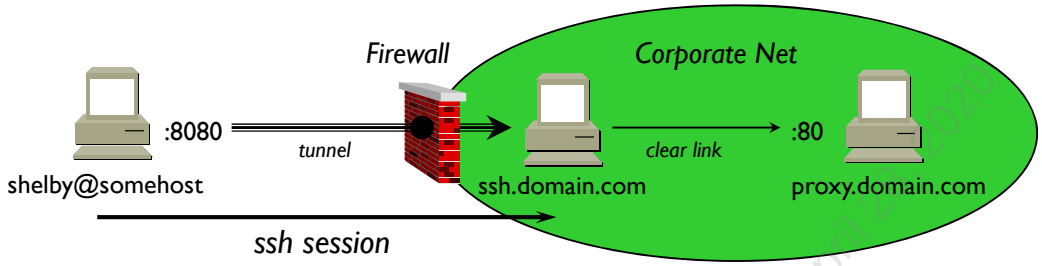
The solution is to set up port forwarding to a single internal HTTP proxy server

It's certainly possible to use SSH tunnels to access a remote web server behind your firewall in much the same way we accessed an IMAP server in our earlier example. The problem you run into, however, is when you have to follow a link off of the web server you're browsing to some other internal web server. Unless you have a tunnel set up to access that other web server, the link breaks. Actually, the link will probably break even if you have the tunnel set up because the link is going to tell you to go to "`otherwebserver:80`", and not "`localhost:8888`" or whatever port you have the tunnel set up on.

What you'd really like is a "seamless" mechanism for being able to "see" and browse all internal intranet websites within your organization's networks. The trick here is to not set up port forwarding to individual web servers, but instead to set up a port forwarding channel to an internal HTTP *proxy server.* This way, you can proxy all of your HTTP requests to this internal server, which can "see" your entire internal intranet because it sits *behind* your firewalls on the other end of the SSH tunnel.

HTTP Proxy Solution

ssh –L 8080:proxy.domain.com:80 ssh.domain.com



- Set "HTTP Proxy" in browser to "`localhost:8080`"
- Joe can now "see" all internal websites
- Also, can "protect" Joe's external browsing

The picture above describes this architecture. We SSH into our internal SSH gateway machine, ssh.domain.com, and set up a port forwarding channel between localhost:8080 on your local machine outside the firewall to port 80 on the proxy server inside of your corporate firewall envelope. Now go into your web browser settings and set the HTTP proxy configuration to "localhost:8080". All of your future HTTP requests will follow this setting down the SSH tunnel to the internal proxy server, where they will get relayed to the appropriate internal web server. You now have total visibility to your company's intranet.

What's interesting about this configuration is that all of your local web browsing is now going down the SSH tunnel to the internal proxy server. So, for example, if you surfed to a website like Google, that HTTP transaction would go to the internal proxy server and then back out to Google via your regular corporate firewall. While this is probably going to make your web access a little bit slower overall, it has some distinct advantages.

The most significant advantage is that your web browsing is now "protected" by whatever content filtering you have set up at your corporate internet gateway to protect users from malicious active content or inappropriate material, and you can also take advantage of web cache servers you may have set up for your corporation. Also, remote websites will never see the "real" IP address of your local machine, since all HTTP requests will apparently be originating from your internal corporate network (or whatever NAT gateway you're using to reach the internet) rather than your local system.

Other Proxy Ideas

SOCKS is a generic application proxy

- Tunnels TCP and UDP using special "SOCKS-ified" clients
- "Dynamic port forwarding" in OpenSSH ("**`ssh -D …`**")

OpenSSH 4.3 added full VPN capability

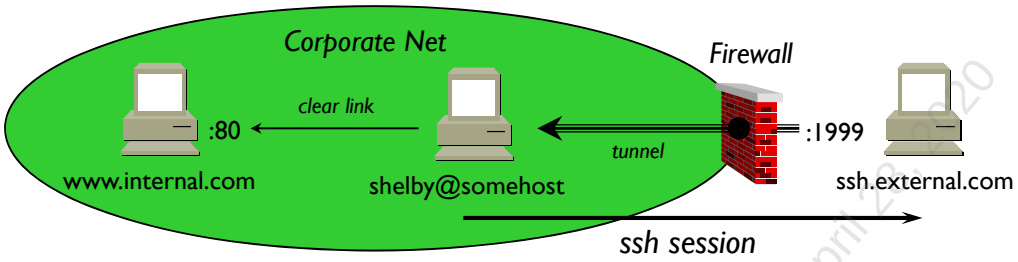
The previous example works only for web traffic, but you can use a similar configuration with other more generic proxy systems. SOCKS is a generic proxy solution that uses specially "SOCKS-ified" network clients that communicate with a SOCKS proxy server, which handles all of the normal application protocols with the remote server you actually want to talk to. The SOCKS clients talk to the SOCKS server over a single TCP port (1080/tcp by default), so this traffic can be tunneled via SSH. More on SOCKS at *http://en.wikipedia.org/wiki/SOCKS*

OpenSSH has SOCKS support built into the client and server. You can use "`ssh -D <port> …`" to set up a local port forwarding that allows SOCKS clients to connect to this port and proxy SOCKS connections through your encrypted SSH connection to the remote network. The SSH server on the remote network will act as a SOCKS proxy server automatically, so you don't have to manually configure a SOCKS server.

OpenSSH 4.3 (Feb 2006) added support for true layer 2/3 VPN activity over an SSH session. See the "`-w`" option in the `ssh(1)` manual page.

Reverse Port Forwarding

ssh –R 1999:www.internal.com:80 ssh.external.com



- User has just subverted corporate policy
- *ssh.external.com* can now route to internal web resources
- Not detectable (except from Shelby's machine)

This is the one that's going to keep you up at night …

So far, we've only been looking at regular port forwarding with SSH where the user sets up a tunnel from their local machine to some internal resource. However, SSH also supports what's called *reverse* port forwarding, where one of your internal users sitting behind the firewall can SSH out to some external system and set up a tunnel endpoint on that external machine that gives access to your internal corporate resources behind the firewall. That means anybody with an account on that external system can potentially access your internal resources. Actually, it could be even worse than that because the remote SSH server may have the "`GatewayPorts`" option enabled, which means that the tunnel isn't just bound to "`localhost:1999`" (per this example), but is in fact bound to port 1999 on *all interfaces* on the remote system, giving access to your internal resources to anybody who can route packets to the external machine.

Remember that from an external network monitoring perspective, there's no way to detect that this is happening—all of the tunneled traffic is happening over the encrypted SSH session. Sure, if you happened to be on Joe's machine, you could see what arguments he used with the SSH client application, but in a large network, it's not practical to monitor every SSH invocation in this fashion.

Blocking outgoing SSH at your firewall probably isn't going to help. If you block port 22 but still allow HTTP (80/tcp) or HTTPS (443/tcp) traffic, then a sufficiently motivated user is simply going to set up their remote SSH server to listen on one of the ports that you do allow to escape. If you try and give users SSH clients with the remote port forwarding code removed, users can just download the SSH source and build their own fully functional SSH clients.

Uses for Reverse Forwarding

- Used to tunnel information with legitimate external hosts
  - SQLNet over SSH for internal DB access
  - Syslog-NG via TCP to capture remote logs
- Firewalls typically more permissive "outbound"
- "Inside toward outside" connections are easier

So, at this point, you're probably thinking, "Holy cow! Who the heck thought this was a good idea?" Actually, there are some real-world uses for reverse port forwarding beyond allowing malicious internal users to subvert your firewall policy, because reverse port forwarding also allows your internal administrators to subvert your firewall policy—when it's appropriate to do so for certain critical network traffic.

For example, suppose you have a web farm at some co-location facility that occasionally needs access to some internal database at corporate HQ. You could SSH out from the internal database server on your corporate network to an app server at your co-lo and set up a reverse tunnel so that SQLNet traffic could reach your internal RDBMS as needed. In the next section, we'll be using reverse port forwarding to allow remote `Syslog-NG` servers to talk to an internal central log server on our corporate network.

The point here is that most firewalls are set up to be more permissive to traffic leaving the corporate network than coming in. It may not be possible for a machine at some remote site or co-lo to SSH into your corporate network and set up a normal SSH tunnel. Reverse port forwarding allows you to set up the SSH session in an "outbound" direction from the perspective of your corporate firewalls and get access to internal data from outside. You just need to be very careful to protect the tunnel endpoint from malicious outsiders.

Syslog-NG

Syslog-NG is an alternate Syslog daemon that can be used to completely replace the vendor-supplied `syslogd` on your systems, or used in conjunction with the vendor-supplied `syslogd`. Syslog-NG has lots of nice features that make it ideal for deployment on central log servers for your enterprise. Syslog-NG has been ported to a wide variety of platforms including Linux, Solaris, HP-UX, AIX, and the *BSDs.

It's worth noting here that there are Syslog agents available for many non-Unix platforms, including Windows. And most routers, switches, and other network devices support logging to a central log server via Syslog. So, you really can use Syslog as the "lowest common denominator" application protocol for creating a centralized logging infrastructure for your enterprise.

Agenda

- Quick Overview
- Installation and Configuration
- SSH Trickery

The Syslog-NG material in this course is divided into three main sections. First, we'll look at why Syslog-NG is useful and some general considerations when deploying Syslog-NG in your environment. Then we'll look at specific configuration examples for setting up a central log server. Finally, we'll go into explicit detail on how to configure two Syslog-NG servers to talk to one another over a reverse SSH tunnel.

Overview

First, some basic thoughts on Syslog-NG and some things you ought to consider before rolling it out in your environment.

Syslog-NG Advantages

- Obvious benefits:
  - TCP-based logging allows SSH tunneling
  - Logging over SSL/TLS
  - More flexible message filtering
- Not so obvious benefits:
  - Creates new log files "on-the-fly," and can use date-based filenames: *auto-rotation!*
  - Can correct deficiencies in vendor-supplied Syslogs (like not listening on 514/udp)

There are a few "advertised" features of Syslog-NG that attract most people initially. First, there's the ability to log messages over TCP—this gives more reliability than the normal UDP-based Syslog logging and allows SSH tunneling. In fact, v3.x Syslog-NG supports logging using SSL/TLS, so you can encrypt the logging streams natively (though it turns out there still may be some use for SSH tunneling—more on that later). Finally, Syslog-NG has the ability to filter log messages in more ways than just normal Syslog facility and priority combinations.

What's less obvious, until you start using it, is that Syslog-NG has some other features not present in your average Syslog. For one thing, Syslog-NG is capable of automatically starting new log files (older vendor Syslogs will not create new log files on the fly, though the Syslog daemon that ships with Linux does create new files on the fly). Also, Syslog-NG allows you to use macros like the date and time in log filenames. So, if you set things up properly, you can have Syslog-NG automatically create new log files every day. This means that you generally don't need some other external "log rotation" program.

Syslog-NG also gives administrators more control than many vendor-supplied Syslog daemons. For example, on many older, proprietary Unix systems, it may not be possible to prevent the vendor-supplied Syslog daemon from listening on 514/udp for remote Syslog messages. This can result in "Syslog bombing" as a denial-of-service attack. Since Syslog-NG allows the administrator to specifically set whether or not they want to listen on 514/udp, replacing the vendor Syslog with Syslog-NG can allow you to fix a potential hole in your security configuration.

Syslog-NG also gives you the option of running as an unprivileged user and also running `chroot()`ed, which most vendor-supplied Syslog daemons do not.

Where to Install

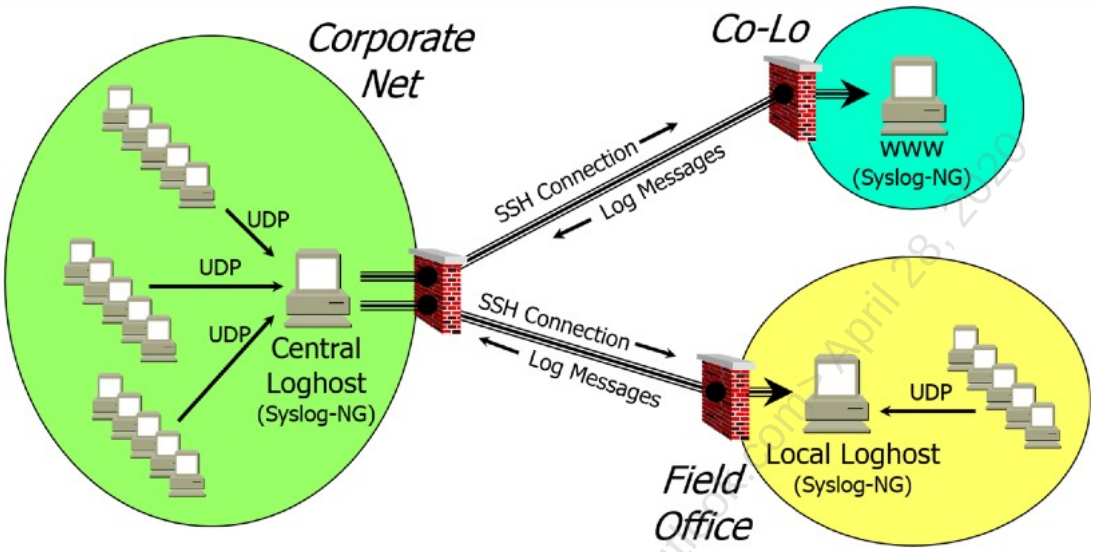
- Run Syslog-NG on your central network log server
- Use Syslog-NG over SSH through firewalls
- May not want to replace the Syslog daemon everywhere
  - Too much administrative hassle
  - Syslog-NG server can handle normal UDP syslog messages

While SUSE and Gentoo Linux ship with Syslog-NG as their default Syslog daemon, this is not true for most Unix-like OSes. That means you'll have to install and maintain Syslog-NG yourself in most places you choose to deploy it. You'll definitely want to install and run Syslog-NG on your central log servers, but you may find it to be too much of a hassle to install Syslog-NG on every machine in your environment (too many different software packages to maintain, too much extra administration). Remember that Syslog-NG is perfectly capable of accepting remote Syslog messages from "standard" vendor-supplied Syslog daemons via the normal UDP logging mechanism. So you can leave the regular Syslog running on most of your machines and just run Syslog-NG on your log servers.

The only other reason to run Syslog-NG is for the TCP-based logging. If you need to drill log messages through a firewall, particularly over SSH, then you'll need a Syslog-NG server at each end of the connection so that log messages can be relayed via TCP. Now you might have a single log server at a remote site running Syslog-NG that is acting as a "collector" for normal UDP-based Syslog messages from the other machines at that remote location. The Syslog-NG server can then relay those messages via TCP to a central log server on your primary enterprise network.

It may be easier to understand all this by looking at the network diagram on the next slide …

Sample Architecture



On the left side of the picture, we have our primary corporate network with a central log server running Syslog-NG. This server may be accepting standard UDP Syslog traffic from dozens or even hundreds of machines at this site. The basic limitations here are the amount of network bandwidth to the central Syslog-NG server (increase by using multiple network interface cards) and the speed of this machine's disks (increase by using striped RAID volumes spanning many fast disk drives).

From this primary Syslog-NG server, we've also established SSH connections to a couple of different remote sites. These SSH connections are configured with a reverse port-forwarding tunnel so that Syslog-NG machines at each of these remote locations can send log messages via TCP back to the primary log server over the tunnels. At one site, our co-lo, maybe we just have a single web server that's using Syslog-NG to send its logs back to home base (though obviously, it's more likely you'll have a whole farm of web and app servers out there).

But the other remote site is one of our field offices where there are potentially lots of machines. One system is set up as a local log server for the remote field office. This machine collects logs from all of the other systems via the normal Syslog UDP mechanism, and might even save a copy of these logs on its local disk. However, the log messages can also be sent back via the SSH tunnel to our primary Syslog-NG server at corporate HQ for secure, off-site storage and analysis.

Why not just use logging via TLS instead of these SSH tunnels? Well, maybe because the firewall admins at the main corporate location don't want to allow inbound connections (even over TLS) from the remote offices. But the corporate firewall policy most likely already allows the outgoing SSH connections to the remote sites, so reverse tunneling is an expedient way to create the necessary connectivity.

Installation and Configuration

With the general overview stuff out of the way, let's look at some specific information as far as getting Syslog-NG installed and working.

Installation Notes

- Grab Syslog-NG sources *and* dependencies from BalaBit
  - BalaBit's EventLog library is required
  - System Glib version must be recent
- Need up-to-date GNU development environment
- Optionally supports TCP Wrappers, database injects ...

Syslog-NG is distributed for free in source code form from *https://syslog-ng.org/*

In order to build Syslog-NG, you'll need the EventLib library created by the BalaBit developers. Frankly, I wish they'd just bundle this library with the Syslog-NG sources. Also, the Syslog-NG build is particular about the version of the Glib library you're using. The `configure` script will refuse to run, unless you have a particular minimum version number installed. On older legacy systems, it may be necessary to download and install a recent Glib from source. The extra dependency on Glib is a little annoying, but you probably already have Glib installed if (a) you're on a Linux system, or (b) you're compiling lots of other Open Source Software.

The Syslog-NG build assumes that you have all of the standard GNU development tools—`gcc`, `make`, `bison`, `flex`, etc. So, if you're building on a non-Linux platform, make sure you have the GNU stuff you need installed and in your search path.

You can optionally compile Syslog-NG with TCP Wrappers support so that you can filter connections to your Syslog-NG ports via `hosts.allow` and `hosts.deny`. To enable TCP Wrappers support, make sure that you've got the `libwrap.a` and `tcpd.h` files installed (similar to when you compile SSH with TCP Wrappers support) and then use the `--enable-tcp-wrapper` option when running the Syslog-NG `configure` script.

Similarly, support for logging to a back-end database means using `--enable-sql` when running the `configure` script, but will require having the `libdbi` abstraction layer libraries during the build process.

Sample `syslog-ng.conf` (I)

```
@version: 3.2
@include "scl.conf"
```

```
options { check_hostname(yes);
          keep_hostname(yes);
          chain_hostnames(no);
};
```

[... continues on next slide ...]

By default, Syslog-NG looks for its configuration information in a file called `/etc/syslog-ng/syslog-ng.conf` (though you can use the `-f` option when starting the `syslog-ng` daemon to specify an alternate config file). Now, some `syslog-ng.conf` files can get pretty complicated, but they always contain the same basic sections. I'm using shading here to help visually call out each section so we can focus on one thing at a time.

Starting in the 3.x versions, you now begin your files with the `@version` macro. This is a mechanism to allow backwards-compatibility with config files from earlier releases. The `@include` statement includes the `scl.conf` file that ships with Syslog-NG—this file sets up some macro definitions that we can use later in the configuration file.

The first of the main configuration sections is where you set global options—`options` being a semicolon-separated list of individual options enclosed in curly brackets, with a trailing semicolon after the closing curly brace. The `check_hostname(yes)` option says to verify the incoming hostname on the Syslog message to make sure it only contains allowable characters (alphanumeric characters, hyphen, and period)—this is important since we'll be saving messages in directories by hostname and we don't want attackers putting nasty characters in spoofed log messages that might cause us to overwrite unexpected files. The `keep_hostname(yes)` option tells Syslog-NG to just preserve the original hostname in the message, rather than replacing it with the hostname of the remote IP address the message was sent from—this is important when you're relaying messages through multiple hops, especially if one of those hops is an SSH tunnel (which makes "source" of the message appear to be the loopback address). Setting `chain_hostnames(no)` causes Syslog-NG to present only the original hostname and not the chain of "hops" that the message has been relayed through.

If you flip on over to the next page, we can talk about the rest of the configuration file…

Sample `syslog-ng.conf` (2)

```
source inputs { internal();
                system();
                udp();
                syslog( transport("udp") );
                syslog( transport("tcp")
                        max_connections(100) );
};

destination logpile {
file("/logs/$HOST/$YEAR/$MONTH/$FACILITY.$YEAR$MONTH$DAY"
      owner(root) group(root) perm(0600)
      create_dirs(yes) dir_perm(0700)); };

log { source(inputs); destination(logpile); };
```


The next section is where you make "`source`" declarations—where Syslog-NG going to receive incoming messages from. The `internal()` definition means messages from the Syslog-NG daemon itself and *not* messages from other local processes on the system. Local logging is handled by the `system()` directive. It turns out that `system()` is actually a macro that's defined in the `scl.conf` file we talked about on the last page. The `system()` macro gets replaced at runtime by the appropriate configuration statements (based on the OS platform that Syslog-NG detects that it's running on) to handle receiving local log messages. This is a huge improvement over earlier releases (prior to Syslog-NG v3.2) where administrators manually had to configure these rules for each platform (and every Unix-like system does this differently).

The `udp()` channel tells Syslog-NG to listen for traditional (BSD-style, aka RFC 3164) Syslog messages on 514/udp. You'll probably want to enable this for backwards-compatibility with traditional Syslog devices. There's also a `tcp()` option that listens for BSD-style Syslog on 514/tcp.

But BSD-style Syslog logging is being deprecated in favor of the newer RFC 5424-style logging. This newer style of logging has many advantages—a more structured message format that is easier to parse, plus standard GMT timestamps that make cross-timezone logging a lot easier to deal with. So, `syslog(transport("udp"))` means to listen for RFC 5424 logging on 601/udp (the default port for this protocol). Similarly, `syslog(transport("tcp"))` does the same thing on 601/tcp. However, with TCP-based logging, it's important to remember that TCP log connections are *persistent*—meaning that if a host starts logging to your server via TCP, it will continue to hold that connection open, even if no logs are currently coming through. So, it's important to increase the maximum number of simultaneous connections your Syslog-NG daemon can handle (the default is only 10, which is not normally enough for a heavily used server). Typically, each input type has many other options that can be set—like alternate port numbers to listen on, etc. Consult the Balabit documentation for more info.

The next section in the configuration file is where you specify different "destination" blocks—places where log messages might end up. Like the regular Syslog daemon, Syslog-NG is capable of putting messages in files, sending them over the network to other systems, or even sending messages to specific users' ttys. Here, we're defining a `file()` type destination. Notice the use of built-in Syslog-NG macros so that we can put messages into files based not only on the Syslog facility but also by date and even by hostname. Unlike older Syslog daemons, Syslog-NG will create new files as needed, and we specify the ownerships and permissions for new files as options within the `file()` directive. We also tell Syslog-NG to create new directories as needed and what permissions to set on the new directories.

Having made one or more source and destination declarations, you need to combine them with "`log`" directives to actually make logging happen. Here, we've got a very simple declaration that just takes all of our incoming log messages and dumps them into our one destination; but, of course, that `destination` directive causes the log messages to be automatically split into potentially hundreds of different files.

Frankly, the basic configuration shown here works perfectly well for most central log servers. It may be that you don't ever need to do anything else than what you see here. But you may need more complicated configurations to deal differently with "local" and "remote" Syslog messages if you're setting up a log server at a remote location. And we haven't even touched Syslog-NG's filtering capability. So, let's investigate all of this in the next several slides.

Adding Complexity

```
source local  { internal(); system(); };
source remote { udp();
                syslog( transport("udp") );
                syslog( transport("tcp") ); };
```

```
destination logpile { … };              # per previous example
destination off_site {
    syslog("loghost" transport("tcp"));
};
```

```
# Local messages to local log files and off-site
# Remote messages simply forwarded to off-site loghost
log { source(local); destination(logpile); };
log { source(local); destination(off_site); };
log { source(remote); destination(off_site); };
```


Here's a partial `syslog-ng.conf` file example that demonstrates the power of using different sources and different destinations.

First, we have our "local" sources: Syslog-NG logs messages–`internal()` and `system()`. Then there are "remote" messages received over the network from other machines nearby. Similarly, we have one destination that is sending messages to local files (per our previous `file()` example) and another destination that's sending messages via TCP to some remote log server.

Now, we use log statements to handle our local and remotely sourced messages differently. We should keep a local copy of messages generated on the local machine ("`source local {…};`"), but we also want to send these messages off-site to our central log server. However, we don't care about storing messages received from remote systems on the local machine. We just want to relay these messages down our TCP connection to the remote log server for storage on that machine (or possibly that machine relays them on to some even more distant log server).

This is the kind of configuration you might use on that log server at the remote field office in the network diagram we saw earlier. Though I suppose it's possible in that case that you might want to keep a copy of the logs received from other machines at the site locally on this log server ("`log { source(remote); destination(files); };`") as a redundant copy and for local administrators to refer to.

Having a local copy of the logs also helps in case there's a network outage. Syslog-NG has some built-in log buffers that it will use to store messages that should be sent to the central log server. But in an extended network outage, Syslog-NG will run out of buffer space and have to start throwing away the oldest log messages (the size of the buffers is tunable, but there are practical limits). If you have a local copy of the logs on the disk of the remote log server, you at least have the option of manually copying the log data to the central log server after the network outage is over.

Filters

Filters can select specific messages by:

- Standard Syslog facility/priority values
- Program name or host name (regexp match)
- Regexp match against message contents

Logical operators (and, or, not) also work

Don't try to emulate "old" Syslog configs using filters—change your mindset!

So far, we've gotten a lot of mileage out of the basic Syslog-NG configuration directives without even resorting to specialized filtering, but for the sake of completeness, let's look at some filtering examples.

As you might expect, Syslog-NG allows you to filter messages based on the simple Syslog facility/priority values that normal Syslog daemons allow you to use. However, Syslog-NG also allows you to do filtering with (`egrep`-style) regular expressions against the program name and/or host name in the message as well as against the contents of the actual message. Multiple specific filters can be combined using the normal logical operations: and, or, not.

Before we go any further, let me make it clear that it's necessary to adjust your mindset about routing Syslog messages when you move from the standard Syslog to Syslog-NG. In the standard Syslog universe, you're used to configuration directives like:

```
*.err;kern.debug;daemon.notice /var/adm/messages
```

However, it's clear that Syslog-NG was designed to use a `file()` directive like the one shown earlier to simply split messages into different files, one per each Syslog facility.

So, you typically don't use Syslog-NG filters to "emulate" your original vendor `syslog.conf` file. Instead, filters are generally used to pull out "interesting" traffic into special log files. For example, there's a worm running around your network that creates a certain log signature. You can use a filter to pull out these messages and put them in a special log file so you can easily see which machines are infected and go fix them.

Emulation Example – Don't!

```
# Emulates the following syslog.conf line w/ Syslog-NG
# *.err;kern.debug;daemon.notice          /var/adm/messages

filter kernel { facility(kern); };
filter daemon { facility(daemon) and level(notice..emerg); };
filter err_up { level(err..emerg) and
                not facility(kern,daemon); };

destination messages { file("/var/adm/messages"); };

log { source(src); filter(kernel); destination(messages); };
log { source(src); filter(daemon); destination(messages); };
log { source(src); filter(err_up); destination(messages); };
```


To drive home the point on the previous slide, let me show you an example of what it would take to get Syslog-NG to emulate our hypothetical "`*.err;kern.debug;daemon.notice  /var/adm/messages`" example. What's one line in a normal `syslog.conf` file actually converts to the *seven* separate declarations shown on the slide above (and remember the typical `syslog.conf` file has potentially a dozen or more lines in it).

First, we define a simple filter that matches any message sent to the `kern` facility—this corresponds to the "`kern.debug`" portion of our example, meaning match all `kern` messages of debug priority (the lowest) and higher. Next, we have a filter for the "`daemon.notice`" portion of the `syslog.conf` expression, which matches the `facility(daemon)` and the priority levels from `notice` upwards. Notice the use of a range specifier there ("`notice..emerg`"). Finally, we need to deal with the "`*.err`" portion of the `syslog.conf` example, so we match priority levels "`err..emerg`" for everything *except* the kern and daemon facilities we dealt with in the other rules. Then, of course, you need a `destination` declaration that sets up the `/var/adm/messages` file.

Once you've got the filters and destination set up, you can actually start creating `log` directives (I assume the `options` and `source` directives are set up elsewhere in the file). When you use `filter()` expressions inside a "`log {…};`" directive, the particular message must match *all* of the defined `filter()`s in order to be logged to the `destination()`. So, we actually have to set up three separate log statements to make sure each type of message we want ends up in `/var/adm/messages`.

So, the point is that a regular `syslog.conf` file lets you define very specific kinds of things very tersely, but doesn't give you much flexibility. Syslog-NG gives you lots of flexibility, at the cost of a more verbose configuration syntax. But in any event, don't try and make Syslog-NG work just like your old Syslog configuration—embrace a new mode of logging instead.

More Interesting Use of Filters

- Solaris et al use older Syslog conventions: "cron" facility number is 15, not 9
- Syslog-NG built on Linux doesn't recognize the old facility number
- Result is filenames like *"F.YYYYMMDD"* showing up in your log directories
- Can use filters to redirect msgs correctly

Now let me show you a worthwhile use for Syslog-NG filters. This is an example of a workaround that's needed when you've got a network of both newer Linux and *BSD systems as well as older, proprietary Unix systems like Solaris and HP-UX boxes. In this case, we'll use the example of a central Syslog-NG server running on a Linux machine, receiving logs from Solaris and HP-UX systems, but the basic gist of the example is equally valuable when you're going the other way as well.

The basic problem is a disagreement between the "new" Syslog "standard" used by Linux and *BSD and the "old standard" used by Solaris, HP-UX, etc. On Solaris machines, the `cron` facility is facility number 15, whereas it is facility number 9 on newer systems. So, what happens when you Syslog messages from a Solaris machine, the Syslog-NG server on Linux doesn't recognize the facility number. If you tell the Syslog-NG server to save these messages in a file according to the `$FACILITY` macro, the Solaris Syslog-NG server will just plunk the `cron` messages into a file named "`F`" (the literal facility number in hex). This is not what you want.

Luckily, you can use filters to match these "unknown" facility values and route them into the proper files. Check out the example on the next slide…

Work-Around for Solaris

```
filter old_cron { facility(15); };

destination cronlogs {
     file("/logs/$HOST/$YEAR/$MONTH/cron.$YEAR$MONTH$DAY"
          owner(root) group(root) perm(0600)
          dir_perm(0700) create_dirs(yes)); };

# … other destination not shown for sake of clarity …

log { source(src); filter(old_cron); destination(cronlogs); };
log { source(src); destination(logpile); flags(fallback); };
```


OK, first we have to write `filter` rules that match facility number 15. For Syslog-NG v2.x and later, this is straightforward as you can see in the example above.

Now if everything were working perfectly, all `cron` messages would end up in files called `cron.`*YYYYMMDD* under our various logging directories. So, we create a `destination` declaration that specifies the actual "`cron`" filename, rather than using the `$FACILITY` macro as we did in our previous examples. We will also need our normal "`logpile`" destination declaration that sorts other messages according to `$FACILITY` (not shown due to space constraints).

So, then we define a `log` declaration to match our facility number filter and get the appropriate messages into the right files. So far, so good. But now we want to have another declaration that handles everything else *except for the messages we've already delivered*—because if you end up logging the old `cron` messages via this default rule, you're still going to get the strangely named files in your log directory. You can add the `flags(fallback)` option to any `log` declaration to only log messages that haven't already been logged via a `log` declaration that does not include the `fallback` option (yes, you can have multiple `log` declarations with `fallback`—maybe one for local and one for remote).

By the way, there's also a `flags(final)` option that does "first match and exit" behavior. If Syslog-NG matches a `log` rule with `flags(final)` defined, then the message is logged and Syslog-NG stops processing other rules.

Other Features

- TLS Logging
- Pipe messages to other processes and into databases
- Reformat log messages on the fly (simplify/standardize)
- Time zone conversions

Syslog-NG has some other interesting features that we don't have time to go into today, but which are documented in the reference manual and FAQ (URLs at the end of this section). For example, I don't have time to fully cover how to set up certificates, etc. for TLS logging. But it's actually fairly easy to figure this out from the reference manual.

One thing that Syslog-NG can do that regular Syslog daemons can't, is to pipe messages to an external program. You could use this to send messages to a real-time monitoring tool. However, prior to v3.x, this was also the mechanism sites would use to shove log messages into a relational database. More recent versions of Syslog-NG have database hooks (that work with MySQL, Postgres, Oracle, and MS SQL Server) which you can call directly in your `destination` blocks via `sql()`. And once your logs are in an RDBMS, you can use relational query tools to analyze your logs. This is very useful if you're dealing with a lot of log information.

Syslog-NG does not have to output messages in the format they were received. You can use output "templates" in your log declarations to output the messages in a more standardized log format to make things easier to parse.

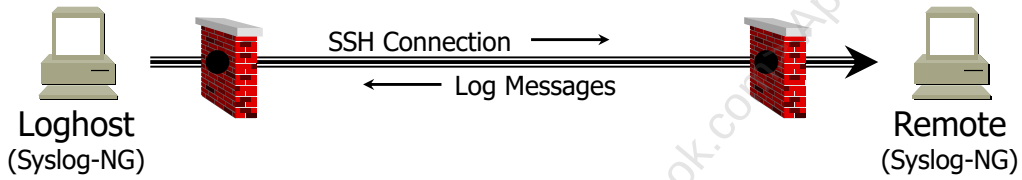
One of the major differences in Syslog-NG 2.x has some protocol extensions to support passing local time zone information between remote Syslog-NG servers and the ability for Syslog-NG to do timestamp conversion on log messages to shift things to the local time zone of the machine where the message is being logged. When receiving messages from a non-Syslog-NG host, the administrator of the central Syslog-NG server can manually define the time zone of the remote host in order to perform time zone conversions, but since this would pretty much need to happen on a host-by-host basis, this seems less useful. Frankly, the time zone support in Syslog-NG seems like a bit of a hack—you're probably better off just running all of your systems in a consistent time zone (like UTC or the time zone of your headquarters) and not bothering to do timestamp conversions. As sites move toward RFC 5424-style logging—which consistently uses GMT for log timestamps—this will become less of an issue.

Syslog-NG Over SSH

If one of the primary advantages of Syslog-NG is being able to tunnel remote log traffic via SSH, then it's probably a good idea to look at how to do that…

Simple Concept

- Central log server uses Syslog-NG to listen for messages
- SSH connection to remote logger with reverse port forward
- Remote logger also runs Syslog-NG to send TCP via SSH



Conceptually, Syslog-NG TCP logging over SSH is very simple. You will typically already have your central Syslog-NG server set up to listen for remote messages via TCP. The next step is to establish an SSH connection to the remote logging source (again, it's usually easier to make this connection in the "outbound" direction from your internal corporate network where the central log server lives). Use the reverse port forward option with SSH that we discussed in the previous section. Once the tunnel is set up, configure Syslog-NG on the remote system to use the tunnel endpoint as one of its "destination" directives.

Let's take a look at how to do all of this in a more or less step-by-step fashion…

systemd Unit File

```
[Unit]
Description=SSH tunnel to remote.domain.com
After=network.target

[Service]
EnvironmentFile=/root/.ssh/ssh-agent-vars
ExecStart=/usr/bin/ssh -nNTx
     -R 6011:loghost.domain.com:601
     tunneluser@remote.domain.com
KillMode=process
Restart=on-failure
RestartSec=30s

[Install]
WantedBy=multi-user.target
```


We want our SSH connection to start automatically when the central log server boots, and we'd like to ensure that the connection gets restarted if it ever dies for any reason. Happily, we can leverage `systemd` to do both for us. Note the use of the `Restart*` parameters to make sure `systemd` keeps things going for us. That `ExecStart` parameter needs to be *one long line* in the actual unit file—I broke it onto multiple lines here for clarity.

So, what are all those options on the `ssh` command line? Well, `-n` says that the process should use `/dev/null` as its standard input. After all, this SSH process is essentially run as a "daemon" via `systemd`. The `-N` option says to make the connection but don't run a remote shell or another command (we just care about the tunnel), and `-T` says don't try to allocate a tty on the remote machine for the session (no interactive commands). The `-x` option disables X11 forwarding (this is mostly a defense-in-depth measure).

I'm logging into my remote system (`remote.domain.com`) as an unprivileged user called `tunneluser` to reduce my security exposure. Since I'm unprivileged, I use a high port number on the remote end (6011/tcp) and tie it back to 601/tcp on the local system (`loghost.domain.com` in the example above).

For authentication, I'm pulling in the `ssh-agent` environment variables from `/root/.ssh/ssh-agent-vars`. Unfortunately, you can't directly use the output of the `ssh-agent` process as the `EnvironmentFile`. You just want a file that contains the variable settings and *nothing else*:

```
SSH_AUTH_SOCK=/tmp/ssh-Qk9VjMvtUNHj/agent.60994
SSH_AGENT_PID=60995
```

This little bit of command-line kung fu will create the proper file format:

```
ssh-agent | head -2 | sed 's/;.*//' >/root/.ssh/ssh-agent-vars
```

Put your new unit file into `/etc/systemd/system`. If you have multiple tunnels to different systems, just create a separate unit file for each connection. This allows `systemd` to monitor and restart each connection individually, and you to see the status of each individual connection via `systemctl status`.

Once the new unit file(s) are in place, you can `systemctl start` the services to get them up and running. `systemctl enable` will make sure they start at the next reboot. The only issue is our dependency on the `ssh-agent` process. Until an admin manually starts `ssh-agent`, creates the `/root/.ssh/ssh-agent-vars` file, and loads keys, none of our connections are going to work. If auto-start on boot is important to you, you'll need to use key files with null passphrases or some sort of other key store besides `ssh-agent`.

`init` Version

Invoke script from `/etc/inittab`:

```
log1:3:respawn:/usr/local/sbin/tunnel
   -R 6011:loghost.domain.com:601
   tunneluser@remote.domain.com
   >/dev/null 2>&1
```

Script gets `ssh-agent` config:

```
#!/bin/sh
. /root/.ssh/ssh-agent
exec /usr/bin/ssh -nNTx $*
```

SANS | SEC506 | Securing Linux/Unix 271

If you're using an older system that uses `init` instead of `systemd`, we can still accomplish the mission. For `init`, you start the `ssh` process from the `/etc/inittab` file with the `respawn` option.

The trick is to invoke a script rather than invoking `ssh` directly. The advantage to calling a script is that the script can first acquire the `ssh-agent` configuration variables and then `exec ssh` with the appropriate arguments. It's important that our script calls `exec` here—we want our script process to be replaced by the `ssh` process so that `init` isn't confused into spawning too many copies of the script.

Note that, in this case, the `/root/.ssh/ssh-agent` file should be the raw `ssh-agent` output to be compatible with our shell script.

Syslog-NG on Central Loghost

Have at least one input group that's listening on TCP:

```
options { keep_hostname(yes); … };
source remote { udp(); syslog(transport("tcp") …); };
# … blah blah blah …
log { source(remote); destination(logpile); };
```

***keep_hostname(yes)** option is critical!*

On the central log server, you must be listening on TCP for incoming log messages and have a log declaration that's putting these messages someplace. Again, you probably have this configured on your central log server already.

As I mentioned earlier, the `keep_hostname(yes)` option becomes particularly important when you're logging messages over a tunnel like this. As far as Syslog-NG is concerned, the messages from the tunnel will be originating from `localhost`, so you wouldn't want Syslog-NG to overwrite the original hostname on the message.

Syslog-NG on Remote Logger

Remember that tunnel will be bound to "localhost:6011":

```
destination local { file(…); };
destination remote { syslog("localhost"
                            transport("tcp")
                            port(6011)); };
log { source(src); destination(local); };
log { source(src); destination(remote); };
```

Always keep local copy of logs!

On the remote log server, you need to define a destination that uses `syslog("localhost")`, which is the remote tunnel endpoint. However, the default port number for Syslog-NG's `transport("tcp")` directive means that you'd be logging to `localhost:601`, which is not what we want for our example. So, we add the "`port(6011)`" option to bind to the alternate tunnel endpoint we set up as our unprivileged user in the previous SSH example.

If the remote log server is simply acting as a relay for log messages from other systems (like at the "field office" in our earlier example), it's up to you whether you want to keep a local repository of those messages on your remote Syslog-NG server. If you have administrators on-site at the remote location, they'd probably appreciate a local, centralized copy of the log messages from the various machines at that site.

Again, the other reason to keep a local copy is in case you have a prolonged network outage. When the outage is over, you can make a copy of the local logs on the central log server by hand.

Is it Working?

- Make sure all the pieces are running:
  - Do **`systemctl start`** for SSH tunnels
  - (Re)start Syslog-NG on both ends
- Logs should now be flowing—check to see if logging directories get created
- You can always do testing with **`logger`** command on remote log source…

Once you've got everything configured properly, do a `systemctl start` for each tunnel on the central log server so that the SSH connection gets started (you should be able to see these SSH processes when you do a `ps`). The equivalent on `init`-based systems would be `kill -HUP 1`. Then `HUP` or restart Syslog-NG on both ends to make sure the TCP logging is going.

At this point, you should be getting log messages from the remote system. If you look in your log directories, you should see new directories have been created for the remote system(s) and new log files should be appearing. To validate that traffic is flowing, you can always run the `logger` command on the remote system to generate Syslog traffic:

```
logger -p auth.info Syslog-NG remote logging test
```

If you've got everything right, the "`Syslog-NG remote logging test`" message should show up on your central log server in the file containing "`auth`" messages from the remote server—in our previous example, this was `/logs/$HOST/$YEAR/$MON/auth.$YEAR$MON$DAY`.

Wrapping Up

- Give up "old" Syslog mindset when using Syslog-NG
- Still need other tools, however:
  - Log analyzer/reporter
  - Tool to compress old log files (see notes)
- Also, the question of how long to keep log files...

The power and functionality of Syslog-NG make it amazingly useful, particularly for large, central log servers in your environment. But to really get into Syslog-NG, you have to be willing to let go of the "old" Syslog configuration practices you've already learned.

But now that you're collecting and storing all of these logs, you're going to need to control their growth. So, while Syslog-NG doesn't require an external log rotation program, you might want to run a simple `find` command through your log directories that just compresses any log files more than a week old:

```
find /logs -type f -mtime +7 -exec gzip {} \;
```

At some point, you're also going to have to decide when to delete log files. If you have a large disk array for your log server, you may be able to keep logs going back for years. With less disk space, you need to remove old log files more quickly. In general, you'd like to hold onto logs as long as you possibly can, just so you can go back and refer to older logs if it takes you a while to spot a problem. Some sites have very specific log retention policies (DoD sites require five years of backlog archives in my experience).

Now there's not much point in collecting logs if you never look at them, but reading log files is a truly tedious business. Smart administrators will use a program to filter their log files and only report "interesting" events.

Quick Notes on Log Analysis

- Do it ... or FAIL!
- Throw away "uninteresting" and look at everything else
- Arrange daily end-to-end test of logging infrastructure

OK, you've now arranged to collect all of your logs at a central log server. But if you never look at the logs (except perhaps after a catastrophe), why did you go through all that work to collect them in the first place? Your logs can alert you to plenty of problems—not just security issues, but basic "block and tackle" kinds of IT issues, like "Hey, the third drive in our RAID array is dying. Maybe we should order a new one." If you're not looking at your logs, you're just living in denial.

Of course, human beings are bad at boring, repetitive tasks like looking at log files, so you're better off having a program keep an eye on your logs and just tell you when something "interesting" happens. The question is how do you get to the "interesting" stuff? In my opinion, the right way to do this is to give the log analysis program patterns that match and throw away the stuff in your logs that you know is *not interesting* and look at everything else. Now some of the "everything else" may be uninteresting messages that you've just never seen before. In which case, you tune your filters to throw away these messages as well. But there's a bunch of stuff left over that's going to be interesting in various ways—some of which you also may have never seen before and never would have known to write an explicit filter to catch. That's the crux of the issue; if you try to write filters that are designed to catch the interesting stuff, there will always be interesting messages you never knew about and wouldn't know to write filters for. So, the only sane approach is to get rid of the junk and look at everything else.

Another classic problem with log analysis is that "no news may be bad news." If you're not getting any alerts from your log monitoring program, it could be because there's nothing on fire. But it's more likely that something has broken down somewhere, and it's preventing you from seeing the alerts. Maybe your central log server has stopped getting log messages, maybe your email system is broken and the alerts from your log monitor are stuck and unable to get to you, and so on. So at least once a day, be sure to generate a spurious alert message that will trigger an alert from your log monitoring system. If you don't get this alert, then you'll know something has gone wrong.

URL Summary

Homepage:
https://syslog-ng.org/

Reference Manual:
[see notes]

FAQ:
http://www.balabit.com/wiki/syslog-ng-faq

The reference manual can be found at:
https://www.syslog-ng.com/technical-documents/doc/syslog-ng-open-source-edition/

Lab Exercise

- Configuring Syslog-NG
- Practice some of the configuration details from class
- You'll need to complete this exercise to be ready for the capstone tomorrow ...

Exercises are in HTML files in two places: On the course USB media and in the home directory of the SANS user in the lab virtual machine. These two locations contain identical copies of the exercise files—it just depends on whether you prefer to look at them from a web browser inside the lab VM (because you're working in full-screen mode) or from your host operating system.

1. Open your web browser
2. Select "File… Open…" from the browser menu (Ctrl-O is the usual keyboard shortcut)
3. This is Day 3, Exercise 4, so navigate to *.../Exercises/Day_3/index.html* and open this HTML file
4. Click on the hyperlink, which is the title of Exercise 4
5. Follow the instructions in the exercise

Final Remarks

Now for some final thoughts and parting shots.

You WERE Going to Test That, Right?

Small mistakes can result in big security holes–
Test machines before use!

Items to check:

- Network access
- What's running on the system?
- Logging

May also want to run an external vulnerability/port scan...

Over the last three days, we've been talking about a "recipe" for hardening Unix-like operating systems. But good cooks always taste their concoctions before serving them to others. So, it's important that you test the configuration of your systems before you deploy them.

For example:

- You've enabled a host-based firewall, but is it really blocking the right addresses and protecting the right ports? Does it allow access to the networks it should be allowing access to? Is direct root access over the network shut off? Do you see the statutory warning banners when you try to log in?
- Do you recognize all of the processes running on your system? Can any of them be turned off? What about the open ports that you see in the output of `lsof` or `netstat`? Are there listening ports that you can disable?
- Are all of your network activities being logged? What about local logs and audit trails like BSM? Are you seeing your Sudo logs? Is your central log server receiving logs from the machines in your environment? Is your log monitoring tool functioning?

It is also a good idea to run a network-based vulnerability scanner (like Nessus or SARA) against your system to get an external attacker's view of your machine.

Make a Full Backup – NOW!

If your machine is compromised:

- Compare against "gold" version of system
- Rebuild your system quickly

Building new machines:

- "Clone" new hosts by building from backup

If disaster strikes (and it will!)

Now, before you put the machine into production and it has a chance to get hacked, make a complete backup of the entire machine. This is also the appropriate time to generate the initial file integrity database as well as using AIDE or your FIA tool of choice. In the case of disaster or system compromise (when this backup may be your only record of the "normal" state of the machine), you will thank me for this advice.

Once you've got a snapshot of a single secure machine, it's easy to replicate that machine. Simply boot off installation media and escape the install program into a shell. From there, you can format/partition your drives and then restore the backup image to disk. You can manually change the hostname, IP address, name server, etc. or leave that up to DHCP.

Additional Monitoring Ideas

Deploy monitoring agent framework:

- May give you earlier problem warnings
- Centralized console for alerts

Mailing Lists:

- Learn about vulnerabilities quickly
- Apply mitigations prior to patch release

We've talked a bit about different types of logging and monitoring already like Syslog-NG, AIDE, and so on. But there are other types of "monitoring" that you should probably be thinking about to protect your infrastructure.

Nagios (*www.nagios.org*) is a framework that runs arbitrary scripts that check aspects of your system and send alarms if the expected output changes. For example, you could write a script that runs `ps` every so often and sends an alarm if there are extra processes running or to alert you if somebody changes the configuration of the firewall on a system. These alerts are published via a management console that gives you a single place to look to get an overview of what's happening in your network. There are plenty of other similar tools out there, both free and commercial.

Another important aspect of monitoring is keeping an eye on important security mailing lists like BUGTRAQ and your vendor's security alert mailing lists. When new vulnerabilities are announced, it may take vendors several weeks to respond with patches. However, there may be steps you can take in the meantime to mitigate the vulnerability, such as blocking certain ports or disabling a vulnerable service. The trick is to know about the vulnerability as quickly as possible so that you can take the appropriate steps.

SANS publishes its weekly "@Risk" newsletter that does a good job of summarizing new vulnerabilities. See, *http://www.sans.org/newsletters/* for more details.

Other Useful Links

Center for Internet Security
http://www.CISecurity.org/

Lots of good AIX links
http://rootvg.net/

Sean Boran's hardening procedures
http://www.boran.com/security/

Miscellaneous Security Info
http://ist.uwaterloo.ca/security/howto/

The `rootvg.net` site has lots of security links for AIX.

Sean Boran's site has some info on hardening a variety of operating systems and applications.

The University of Waterloo maintains several interesting documents related to security. In particular, there are a couple of documents that look at the various set-UID programs that ship with Solaris and make recommendations on which can have the set-UID bit disabled.

URLs

BUGTRAQ Archives
http://www.securityfocus.com/

Packet Storm
http://packetstormsecurity.org/

Phrack Magazine
http://www.phrack.org/

CERT Advisories
http://www.cert.org/advisories/

The BUGTRAQ mailing list is an excellent source of information on breaking security issues (there's a special NT BUGTRAQ list for Windows-specific issues, though plenty of Windows exploits show up on BUGTRAQ) and general security-related discussion. The archives (with a search engine) and instructions for joining the list can be found at: *www.securityfocus.com*.

Packet Storm is an excellent archive of exploit tools and papers from both the "white hat" and "black hat" communities.

Phrack is the irregularly-published but unusually entertaining collection of serious exploits, snide humor, weird but barely probable ideas, and insane rantings of a parade of odd folks. It's worth reading for the "letters" column alone.

The CERT Advisories paint a nice historical picture of the state of security over the last dozen years. CERT advisories are deliberately vague on actual exploits, but one gets good at "reading between the lines."

Books

Garfinkel and Spafford, Practical Unix and Internet Security, O'Reilly and Assoc., ISBN 1-56592-148-8

David Curry, Unix System Security, Addison Wesley, ISBN 0-201-60640-2

Kernighan and Pike, The Unix Programming Environment, Prentice Hall, ISBN 0-13-937681-X

Nemeth et al, Unix System Administration Handbook, Prentice Hall, ISBN 0-13-151051-7

Chapman and Zwicky, Building Internet Firewalls, O'Reilly and Assoc, ISBN 1-56592-124-0

All of these books figured in the preparation of this course at one time or another.

Garfinkel and Spafford (2003) is sometimes a frustrating book because the authors deliberately don't give full details on attacks and some of their security recommendations can, therefore, appear puzzling. Still, it's a very readable book and a good introduction to Unix security.

Curry's book is even older (circa 1992) than Garfinkel and Spafford. Still, it contains good coverage of "classic" Unix security issues. Also, it has a complete bibliography with a listing of seminal computer security papers.

Kernighan and Pike (1984) is one of the classics in the Unix field. A surprisingly useful introduction to getting around on a Unix system.

The "Nemeth book" is probably one of the best-known books on Unix System Administration. Not long on security information, but a good basic overview of the major Unix facilities from a system admin perspective.

Even if you're only interested in host security, *Building Internet Firewalls* can help you understand the implications of running various network services on your machine.

That's It!

- Q&A
- Please fill out your surveys!

Hal Pomeranz
hal@deer-run.com
http://www.deer-run.com/~hal/
@hal_pomeranz on Twitter

Appendix A

Password Alternatives

This page intentionally left blank.

One-Time Passwords

A One-Time Password (OTP) system guarantees a new password on every connection:

- User establishes secret on remote server
- User has software or hardware "calculator"
- At login, server sends challenge string
- Response calculated with challenge+secret
- Result (password) copied into login window
- Server verifies response

In a One-Time Password (OTP) system, each user still has a reusable secret password. However, this password is used in combination with a special calculator (either a hardware device or a software program) that implements a (sometimes proprietary) cryptographic hash function.

When a user connects to a remote server, the server sends a challenge string to the user instead of the normal `Password:` prompt. The user copies this challenge into their calculator and then types in their secret password. The calculator hashes this information together and computes a response. The user types (or cuts 'n' pastes) the response back to the remote machine. The remote machine knows the user's secret, the challenge it issued, and the hash function, so it can also compute what the user's response should be—thus it compares its result with what the user sent back to verify the user.

Notice that at no time is the user's secret password ever transmitted over the network. Also, since the challenge string varies each time, the same OTP response will never be valid twice. Even if the OTP response were stolen in transit, it does no good to an attacker.

OTPs Are Good, But…

- Attacker can see both challenge and response—brute force attack is possible
- Secret can also be discovered easily by "shoulder surfing"
- Want to prevent access, even if attacker knows our secret…

However, since an attacker listening on the wire sees both the challenge string and the encrypted hash the user sends back, they can try combining the challenge with random password strings until they get an encrypted hash that matches the one the user transmitted. At this point, they will know the user's secret.

The good news is that OTP secrets are typically longer than 8 characters and the hash algorithms used by OTP systems are computationally more difficult than the standard Unix password hash, so it takes a lot longer to brute force OTP secrets. OTP algorithms also generally force users to change secrets regularly (after every 100 authentications or so).

Ideally, however, we'd like to ensure that an attacker couldn't easily break in, even if they were able to deduce the user's secret. This leads us to the concept of *two-factor authentication*.

Two-Factor Authentication

Increases security by combining
- "Something you know" (your secret)
- "Something you have" (physical device)

Physical device is encoded with a unique ID key or algorithm

Most commercial OTP systems employ two-factor auth

Two-factor authentication means that the user must present two proofs that they are who they claim to be. In commercial OTP systems, the two factors are usually:

- A serialized or uniquely keyed hardware device
- A secret password or PIN that the user has memorized

Even if an attacker manages to somehow capture a user's password, they still have to steal the user's personal token in order to successfully impersonate that user. Similarly, if the user loses their hardware token, it's useless to an attacker that doesn't know the user's password. Users must be trained to report lost or stolen tokens immediately, however. The good news is that since users will be unable to log in without their token, reporting tends to be very good.

Types of Two-Factor Devices

Challenge/Response:

- Device is used to calculate responses
- User's secret entered to compute response

Synchronous:

- Continuously produces new passwords
- User's secret "unlocks" token or is combined with password from token

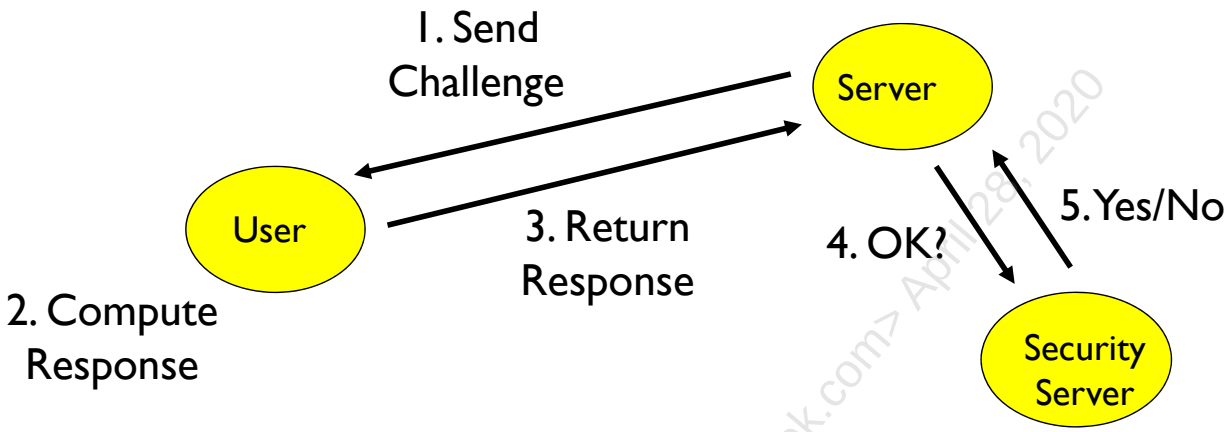
Generally speaking, there are two types of hardware tokens supplied by the various commercial OTP system vendors.

Challenge/response tokens function just like the standard OTP system described earlier, except that the unique identifier or key encoded in the token is used to compute the correct responses. Thus, only the individual who knows the user's secret and has the token with the correct unique identifier associated with that user can compute the correct responses.

Synchronous tokens continuously generate new, unique passwords (typically every 30-60 seconds). The token and the remote machine that the user is attempting to access are kept in sync through some proprietary mechanism (usually involving a trusted third-party "security server" machine). When the user wishes to log in, they must provide the "current" password from their token. The user also has a secret that is also known to the remote machine—sometimes the secret is used to "unlock" the hardware token or permute the "current" password value from the token, and sometimes the secret is merely entered by the user along with the password from the token. Obviously, if the user types their secret over the network, then it is potentially vulnerable to sniffer attacks, but the attacker would still have to steal the user's token.

Users generally hate challenge/response tokens (too much typing) and prefer synchronous tokens.

Logging in with OTPs



Commercial two-factor authentication systems usually require an additional security server machine (typically two or more for redundancy) that holds a database of authentication information that can be queried when a login is attempted. Generally, the process works as follows:

1. The user connects to the remote server they wish to log into via some standard mechanism (e.g., `telnet` or SSH over the network, or via dialup PPP, etc.). They will get a login prompt back from the server, which will typically contain the challenge string if challenge-response type tokens are being used.
2. The user computes the correct response (or simply reads the next password off their synchronous token).
3. The user enters their username and the response they calculated and returns them to the remote server via the normal login mechanism.
4. The remote server then contacts the security server and sends over the user's credentials. This is typically accomplished either via a proprietary protocol or a protocol such as RADIUS or TACACS (for terminal servers and other network devices).
5. The security server verifies the user's credentials and sends back a "yes/no" type response. The security server may also consult its database for other special options (for example, users may only be able to access certain systems at certain times or from other internal hosts, etc.).

Public Key Authentication

- User generates public/private key pair
- User copies public key to remote systems
- At login, remote server encrypts challenge with public key
- User decrypts challenge with secret key
- Decrypted challenge returned to server

Public-key cryptography can be used to do something similar to OTP authentication, and this has been generating a lot of consumer and vendor interest over the last several years.

Each user generates a public/private key pair (messages encrypted with the public key can only be decrypted with the private key and vice versa). When the user wishes to access a machine, that machine must be able to somehow determine the user's public key—usually, this is accomplished by the user simply copying their public key to a file on the remote machine, but could theoretically be accomplished using a trusted third-party certificate authority.

In any event, when the user attempts to log in, the remote machine encrypts a message with the user's public key. The user receives the encrypted message and uses their private key (known only to them) to decrypt the message. When the remote machine receives the cleartext response, it knows the user on the other end of the connection possesses the correct secret key and therefore assumes the user is who they claim to be.

DSA Authentication in SSH is an example of public-key-based authentication.

How to Store Secret Key?

File on disk:

- Must be encrypted to protect secret
- Can still be stolen and cracked off-line
- Not very portable

Smart cards:

- Portable and tamper-resistant
- Need to deploy readers everywhere

The problem with public-key-based authentication is that it is only as secure as the user's private key—anybody who steals that key can impersonate the user until that user "revokes" their old key pair and establishes a new one (potentially a very laborious process).

One option is to encrypt the secret key and store it in a file on disk. Of course, the encrypted file could be stolen and cracked off-line. Also, the file doesn't do the user much good if the user is traveling and can't access the machine that their private key is stored on. Some users put their private keys on floppy disks (or other removable media) for this reason. However, be aware that DOS-formatted floppies have no access control protection, so as soon as that floppy is mounted, any user on the machine can steal the user's (encrypted) secret key file—many users format their floppies as UFS filesystems for this reason (though this means they can't be read on Windows machines).

Smart cards are a portable way of storing and retrieving keys. Generally, smart cards are tamper-resistant—some even going so far as to destroy themselves if somebody appears to be accessing them incorrectly. The major difficulty with smart cards is that they require special card reader hardware to be deployed every place your users might want to log in from (i.e., every computer in the world)—this tends to limit the likelihood of wide-scale smart card deployment in the near future.

At this point, it looks like USB keychain fobs or thumb drives are going to become the standard mechanism for users to carry around their identity certificates—several working implementations already exist. You don't have the "installed base" problem you have with smart cards because nearly every machine these days has a USB interface on it.

How to Distribute Public Keys?

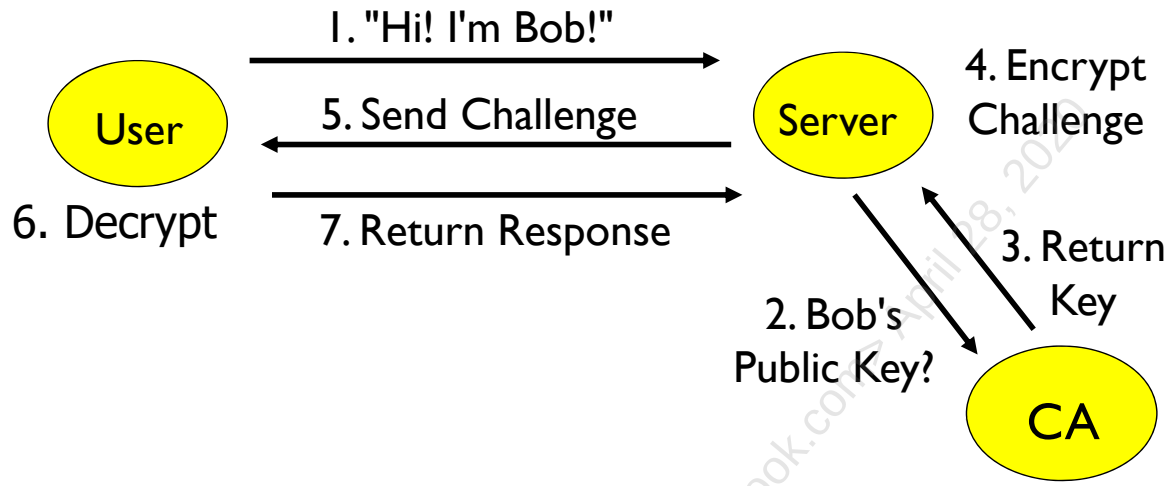
- Each machine or application needs user's public key
- Copying keys is impractical for large networks
- Alternative is a Certificate Authority (CA):
  - Security and trust issues
  - Interoperability issues
  - Who owns the CA? Do they charge?

The other problem is how the server you're logging into is going to get a copy of your public key in order to send you the challenge in the first place.

If you're just using SSH to log into a few machines on the external network of your company, then copying your public key file to each of the machines you need to log into is not such a big deal. Imagine, however, a company that wants to deploy a public-key-based authentication system across the entire organization, or (even worse) wants to interoperate with other organizations in this manner. Expecting users or administrators to copy key files around to thousands of machines is impractical.

The alternative is to establish some central public-key-storage facility that hosts can contact when they want to authenticate a user. This is essentially what a Certificate Authority does. However, if your CA is ever compromised, then you're in big trouble. Also, hosts need to be able to verify in some way that the keys they are receiving are genuine, rather than being inserted by some malicious man in the middle. There are also all sorts of commercial issues to deal with since all of your applications may not interoperate with the same CA or may use different competing key formats. There's also the issue of who "owns" the CA. If this is an in-house-only project, then internal administrators could run their own CA, but global projects generally require some disinterested third party to run the CA. Lots of people are out there already trying to set up competing CAs because of the revenue opportunities (imagine charging $0.10 per authentication!).

Public Key-Based Logins



Let's look at a typical public-key-based authentication session:

1. First, the user enters their username using whatever standard login mechanism is at hand.
2. The remote server now needs to find the user's public key. In some cases, this could be in a file on the local disk, or (as shown above) the remote server might have to go out to a CA to get the key.
3. The CA looks up the user's key and returns it to the remote server. Typically, the CA will digitally "sign" the key with its own certificate in order to help prevent tampering in transit.
4. The remote server now has the user's public key and uses it to encrypt some sort of challenge that can be sent to the user.
5. The remote server sends back the encrypted challenge to the user.
6. The user must decrypt the challenge using their local copy of their secret key (be it in a local file on their disk or residing in a smart card, etc.). Typically, the software the user is using to connect to the remote system handles the decryption transparently to the end user.
7. Finally, the user sends the decrypted response back to the server. The server verifies the response and grants access.

Kerberos

- Secure network authentication system developed at MIT
- Also provides encrypted channels
- Software free, but implementation difficult
- Microsoft implementation only confuses the issue further

Kerberos is an authentication system developed at MIT in the 1980s. While Kerberos is still used at MIT and some other large universities, acceptance in the commercial and government arenas has been slow. Part of this is because Kerberos implementations for large organizations require enormous administrative, development, and user support resources.

Microsoft is now basing its group authentication scheme for Win2K on a form of Kerberos authentication. This, more than anything, may push adoption of Kerberos in commercial institutions. However, Microsoft's Kerberos implementation is incompatible enough from the standard MIT release that you must use Microsoft Kerberos servers (of course). It remains to be seen how successful the Open Source community will be in reverse-engineering the Microsoft Kerberos "extensions."

Aside from providing secure authentication services, Kerberos can also be used to establish encrypted network channels between two hosts.

Principals and Secrets

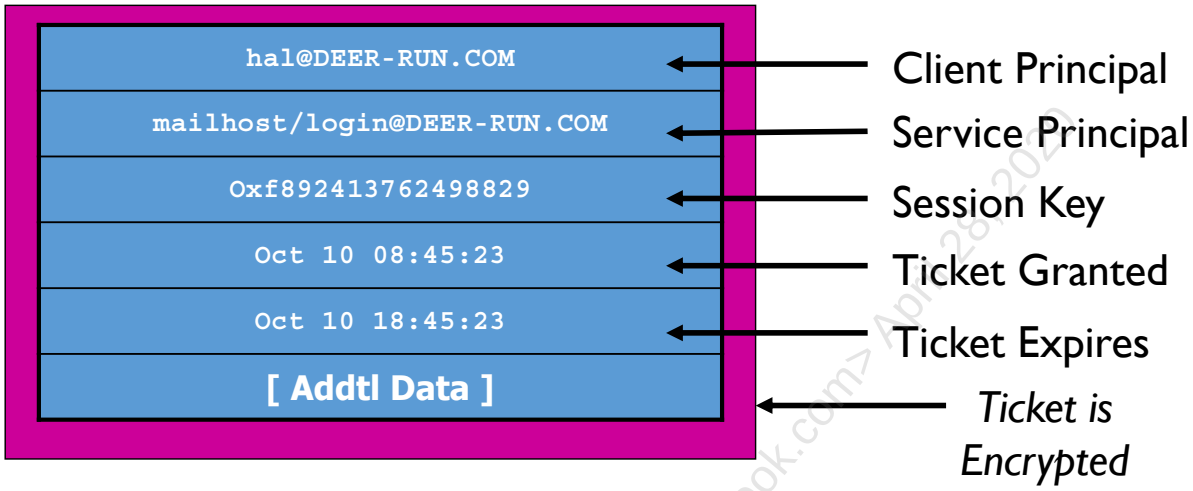
- Users or hosts are called *principals*
- Each principal has a secret key:
  - *User secret is their password*
  - *Machine secrets created by admin*
- All secrets stored in Key Distribution Center (KDC) host
- Hosts have a local copy of their secret

Objects in the Kerberos universe are referred to as *principals*. Generally, these principals are either users or machines. Each principal has its own unique secret key (for users, this is their Kerberos password) which is stored in a central Key Distribution Center (KDC).

A collection of principals is called a *realm*. Kerberos realms and your internal domain name structure tend to be determined by administrative boundaries, so you'll find that most Kerberos installations use domain names as realm names.

Note that Kerberos uses *symmetric key cryptography* rather than public key cryptography—the same key is used to both encrypt and decrypt messages, so if your key is stolen, you are out of luck. This means that protecting your KDC machine is of the utmost importance.

Kerberos Tickets



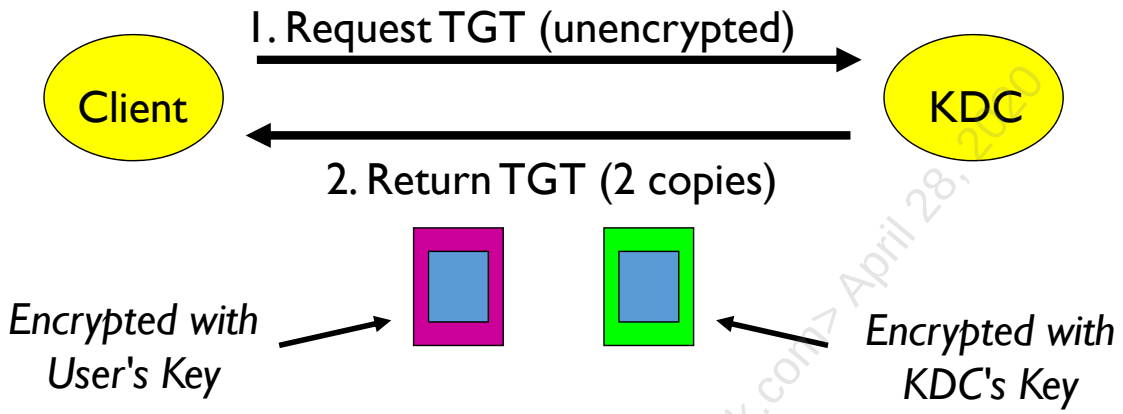
Kerberos uses *tickets* as authentication tokens. These tickets are small encrypted blobs of data that are shipped around the network instead of passwords.

Each ticket contains several pieces of information. The ticket identifies the owner of the ticket (the *client principal*) and the service the ticket is valid for (the *service principal*). In the above example, we see the user `hal` has a ticket for the `login` service on the machine `mailhost` (as opposed to, say, the `POP or IMAP` service on this machine). Tickets are typically only good during a certain window of time and will eventually expire, so they are marked with a *start time* and an *end time*. In order to avoid forcing users to repeatedly reacquire tickets during the day, most sites will set the default expiration time on tickets to a minimum of 10 hours.

The most important element of the ticket, however, is the *session key*. As we will see shortly, this session key is the basis for user authentication in the Kerberos system. The session key is also used to encrypt data between client and server when an encrypted channel is called for.

Finally, the Kerberos ticket contains an area for *additional data*. This field is not generally used in the MIT implementation, but the Microsoft implementation hides all sorts of vendor-specific information in this area.

Getting Your TGT

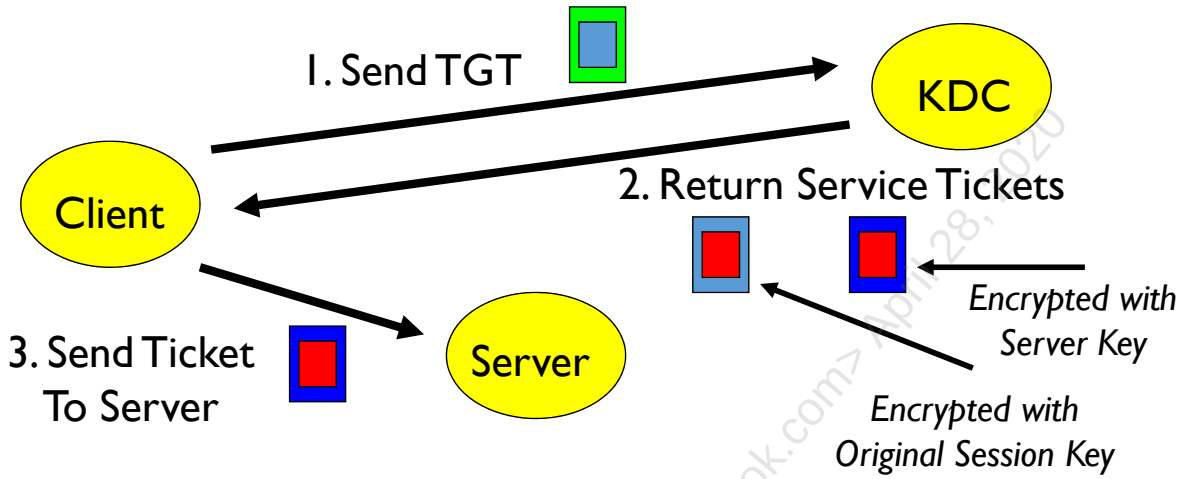


When the user first connects to the Kerberos realm (typically when they first log in during the morning), they receive a *ticket-granting ticket* (TGT) from the KDC. This TGT will be used repeatedly by the user throughout the day (until it expires) to acquire authorization for other services.

The TGT is actually two copies of the same ticket encrypted with two different secrets. One copy is encrypted with the user's secret (password) so the user can decrypt this ticket and extract the session key. The other copy is encrypted with the KDC's key (which the user doesn't have). Both tickets contain the same session key, start/end times, etc.

To find out why this is useful, see the next slide for an example of Kerberos authentication.

Service Authentication



Now let's suppose that our user wants to log into some other machine. They need to acquire a service ticket for the `login` service on the remote server. The client must contact the KDC in order to obtain the service ticket because only the KDC knows the secret for the remote server that the user is trying to access.

The client sends a copy of the user's TGT to the KDC—this is the copy of the TGT that is encrypted with the KDC's secret. The KDC decrypts the TGT and extracts the session key. The KDC then prepares the new service ticket to be sent back to the client—this ticket has its own unique session key and expiration date. The client receives two copies of the service ticket: One encrypted with the session key from the TGT, and one encrypted with the key for the remote server that the user is attempting to log into.

Since the user has their own copy of the TGT with the original session key, they can decrypt the first half of the service ticket they receive from the KDC (the one encrypted with the session key from the original TGT) and see the session key they will use to communicate with the remote server. The client then sends the other copy of the service ticket to the remote server—the remote server can decrypt this ticket since it is encrypted with that server's key. The client and the remote server now share a session key that can be used to authenticate the user (server encrypts a challenge with the session key; only the user should be able to decrypt it), and also as a key for actually encrypting the entire session between client and server (using a symmetric key cipher like RC4 or 3DES).

Problems and Issues

1. Integration and maintenance
2. User resistance
3. Server reliability
4. Time synchronization

Whether you go with a freely available solution or a commercial product, you end up having to replace any software that does regular password authentication with new software, which supports your new authentication solution. This turns out to be a huge number of applications including `telnet`/`rlogin`/`ftp`, SSH, XDM and `xlock`, POP and IMAP servers, Web servers, and proprietary apps like Oracle, Peoplesoft, SAP, etc. Then you get to maintain all of this locally modified software across future revisions of the software itself, the authentication vendor's software, and the underlying operating system. This is an enormous and costly undertaking.

All of these authentication systems are just "different enough" that most users have difficulty accepting them. Also, the use of these systems generally precludes the use of automatic login scripts—security analysts often see this as a feature but users typically disagree. In any event, many sites find it easier to go with a solution that is only deployed at the "edges" of their network—typically only on their dial-in pools and internet/VPN access points. This seems to be a reasonable compromise but can present problems when you are trying to grant a business partner or vendor temporary access to your network.

Generally, all of these authentication solutions are dependent upon some third machine for authentication, be it an OTP security server, a KDC, or a CA. If this machine is unavailable for some reason (whether through accident or malice), nobody is going to be able to log in and get their work done. Your security servers become single points of failure (hint: deploy redundant servers if possible) and prime candidates for attack.

Synchronous OTP tokens and Kerberos are both fundamentally time-based systems. It is vitally important that all of the machines in your authentication realm or domain are keeping the same relative time. However, without external help, system clocks will tend to drift all over the place—this is why we'll be covering NTP later in the curriculum.

Comparing Auth Systems

	Passwords	OTP	Pub. Key	Kerberos
Surfing	X			X
Sniffing	X	+		
Written Down	X			X
Guessing	X			X
Social Eng.	X	X	X	X
Off-line	X		+	+

Let's see how these alternative authentication schemes stack up against standard Unix passwords.

One-time password systems generally prevent most common password-based attacks, though social engineering is always a factor. The most effective social engineering attack is to intimidate an internal helpdesk staffer into subverting the OTP system by claiming to be a VP or the CEO who "lost their token" and "needs to log in right away to close this deal." While two-factor OTP systems are immune to sniffer attacks, software-only OTP systems (such as OPIE or S/Key) can be subverted by sniffing and brute-force guessing as we discussed earlier.

Public-key-based systems are particularly vulnerable to off-line attacks if the attacker can somehow capture the user's secret key file or smart card. Social engineering attacks could also be used to trick the user into divulging their secret key.

Kerberos uses reusable passwords—the passwords just aren't ever shipped across the network in order to thwart sniffers. Attackers are still able to try exhaustive guessing or social engineering attacks to steal the user's password, or the user might write the password down and leave it in a place where it could be found. If the attacker can get the user's TGT, they may be able to decrypt the TGT and extract the session key to impersonate the user. However, the attacker must accomplish this before the TGT expires, so this attack seems unlikely.

506.4-506.5

Linux Application Security Sections 4-5

SANS

THE MOST TRUSTED SOURCE FOR INFORMATION SECURITY TRAINING, CERTIFICATION, AND RESEARCH | sans.org

SEC506.4 & 5 | Securing Linux/Unix

SANS

Linux Application Security

Hal Pomeranz
hal@deer-run.com
http://www.deer-run.com/~hal/
@hal_pomeranz on Twitter

Agenda

- Notes on **`chroot()`**
- The scponly Shell
- SELinux

- DNS and BIND
- Apache

The material in this course book is split over two days.

The first day is devoted to general techniques for securing applications on Linux/Unix. First, we'll briefly look at the `chroot()` mechanism, which can be used to improve the security of many different applications by locking the application up in a small subset of the file system. This may actually prevent certain exploits from working, and will certainly limit the options for the attacker. As an example of a chroot()ed application, we'll look at the highly useful scponly shell, which can be used as a more modern alternative to anonymous FTP. This afternoon, we'll look at the SELinux type enforcement mechanism to further lock down what processes can do on our system.

Tomorrow's material looks specifically at the security settings in popular "internet-facing" applications that typically run on Linux and Unix systems: BIND and Apache. This course is not intended as a tutorial on how to administer these applications. Instead, we'll just be focusing on the "knobs" you can use to improve the security of these applications.

Notes on `chroot()`

`chroot()` is a Unix system call that allows a process to give up access to all but a small portion of the file system. This enables programmers to create processes that run in a captive environment and therefore reduce many of the security risks to applications that have to face "hostile" networks—like the internet.

What Does It Mean to `chroot()`?

- Process calls `chroot()`, specifying a directory argument
- Directory becomes root of private file system for process
- Process unable to access files outside of this directory
- Process is effectively "locked up" in a captive "jail"

When a process calls `chroot()`, it specifies a directory that the process will `chroot()` into. As far as the calling process is concerned, this directory becomes the root of the Unix file system for that process and the process is only able to access files from the `chroot()`ed directory and below. Full pathnames are interpreted as being relative to the `chroot()`ed directory. For example, suppose the process `chroot()`ed to the directory `/local/root` and then tried to access `/etc/passwd`, the process would actually be attempting to access `/local/root/etc/passwd`.

One of the primary uses for `chroot()` is to run networked services in captive environments so that security holes in these services can't be exploited against the entire machine. Typically, these are complex services like FTP, web servers, and DNS servers or services that are insecure in other ways (like the TFTP service that permits file transfers with no authentication).

In the case of a buffer overflow attack or another remote exploit, even if the attacker gets access to the machine remotely, they won't be able to "see" the entire file system to plant their backdoors. In fact, the exploit may fail because of the `chroot()`—if they're trying to `exec("/bin/sh")`, but the `chroot()` directory does not include the `/bin/sh` binary, then the attacker is out of luck.

How Do You `chroot()`?

Some daemons `chroot()` naturally:

- SSH (with PrivSep enabled)
- TFTP
- FTP (anonymous mode)
- BIND (optional)

`chroot` wrapper program can `chroot()` any command

Many daemons `chroot()` automatically, or are at least equipped with the functionality to allow administrators to run them `chroot()`ed if they desire:

- Earlier in the curriculum, we looked at the "Privilege Separation" feature for SSH, where the SSH daemon calls `chroot()` for the potentially dangerous session startup and authentication phases.
- TFTP daemons will `chroot()` to `/tftpboot` (or a similar directory name) so that only files under this directory may be accessed by the TFTP server.
- Similarly, when an anonymous FTP session is started, the FTP daemon will `chroot()` itself to the top of the anonymous FTP directory structure created by the FTP server's administrator.
- BIND (the internet standard DNS server implementation) allows administrators to run the name server in a `chroot()`ed environment, and we will cover this configuration later in the curriculum when we talk about DNS security.

Since many services do not have built-in `chroot()` functionality, most Unix variants come with a `chroot` command that can be used to run any process in a `chroot()`ed directory structure. This is how you typically run applications like Apache in a `chroot()`ed environment.

Why Not `chroot()` Everything?

Requires complete "mini" Unix file systems for each process:

- Binaries and other helper applications
- Shared libraries
- Devices
- Configuration and log files

Programs with too many dependencies difficult to `chroot()`

The problem with running applications in a `chroot()`ed environment is that the environment itself is often difficult to set up initially. Remember that the process is completely trapped in the `chroot()`ed directory—all binaries, shared libraries, system devices, configuration files, etc., must be properly configured by the administrator in the `chroot()`ed hierarchy before the application can be run. Since the application gives few clues about which files and devices it needs to operate (other than failing to run if everything isn't set up exactly right), this process is fairly tedious and complicated.

It's not uncommon to find applications that are so deeply intertwined in the operating system that it's effectively impossible to run them in a `chroot()`ed environment. These sorts of applications should then be allowed to run by themselves on "sacrificial" machines where any security breaches can be more easily contained.

Only `root` Should `chroot()`!

- Attacker sets up `chroot()` area in their home directory
- Creates local password file with known root password
- Runs `chroot` on `su`, now `root` in `chroot()`ed area
- Attacker creates a set-UID shell here
- Attacker now has root shell in home directory!

The `chroot()` system call is only allowed to be called by processes running as root. If normal users are allowed to `chroot()`, then they can exploit this call to gain superuser privileges. For example, there's the attack that was documented in CERT Advisory CA-91.05 (see *`http://www.cert.org/advisories/`*) and affected DEC machines running Ultrix.

Apparently, the Ultrix developers felt that is was useful to allow any user to run the `chroot` program on their processes, so they made the program set-UID root. On the face of it, this sounds like a great idea—users can run suspicious programs in a `chroot()`ed jail and enhance the overall security of the system.

The problem is that a malicious user can exploit `chroot` to get a set-UID root shell. All the user has to do is set up a `chroot()`ed environment that contains a password file they've generated with a known root password, plus a copy of one of the Unix shell executables. The user then runs `su` (or some other similar program) in the `chroot()`ed jail. The `su` program looks up root's password in the dummy password file and the attacker is able to log into the `chroot()`ed area as root. At this point, the attacker sets the set-UID bit on the shell executable in the `chroot()`ed area and then logs back out. The attacker now has a set-UID shell they can use to compromise the entire system.

The "fix" is to simply never set the set-UID bit on the `chroot` program and to never allow normal users to call the `chroot()` system call.

Also Look Out For …

- Attacker exploits poor configuration, and manages to upload a set-UID program into `chroot()` zone
- Attacker can then trigger set-UID program in system environment not protected by `chroot()`:
  - Log in as normal user and execute
  - Buffer overflow in unprivileged program
- Could use this privilege escalation to escape `chroot()` (see upcoming slides…)

You also need to be careful about attackers being able to upload files, particularly set-UID executables, into your `chroot()` area. A set-UID executable in the `chroot()` area could be combined with an unprivileged exploit to gain remote root privileges.

For example, if your anonymous FTP server allows file uploads and doesn't reset permissions on uploaded files, you might be allowing attackers to upload set-UID programs into your upload area. Now, suppose you were also running a web server on the same machine (generally a bad idea for exactly the reason we describe here). The attacker might have a buffer overflow exploit against the web server. Normally, this would give the attacker access as the UID the web server was running as, but if the attacker can have their exploit trigger the set-UID binary they uploaded into your FTP area, then the attacker can possibly get root privileges on your machine!

Similarly, if you allow user logins on your anonymous FTP server, then one of your users might upload a set-UID program into your FTP area and then log in under their own unprivileged UID to trigger the exploit. Later in the class, we will show you good ways to configure your anonymous FTP server to prevent this sort of thing from happening.

Also, it turns out that an attacker who manages to get root privileges while in the `chroot()` area may be able to "escape" from the `chroot()` restrictions. Let's look at this issue in more detail in the next few slides…

Give Up `root` Privileges!

Process must run as superuser in order to `chroot()`

Process *must* give up root privs ASAP, or escape possible:

- Attacker creates devices in `chroot()` area
- Attacker uses `ptrace()` on other apps
- Attacker loads kernel modules

As we've already described, `chroot()` is a privileged system call and can only be run by the superuser. However, after the process has called `chroot()`, it's important that the process give up root privilege as soon as possible and operate as a normal unprivileged user. If the process is running with root privilege and the attacker was able to execute arbitrary code (via a buffer overflow or some other attack), then even a `chroot()`ed process running as root can do a lot of damage.

For example, the attacker could create copies of the system's disk devices in the `chroot()`ed area and have access to the file system (or a copy of `/dev/kmem` to access the kernel). The `ptrace()` system call would allow the attacker to control other processes on the system—i.e., ones that are not locked up by `chroot()`, but instead, have access to the entire system. Or the attacker could exploit loadable kernel modules and put code directly into the kernel (assuming they could get the malicious kernel module into the `chroot()` area).

This is one reason why anonymous FTP servers run as the `ftp` user and why TFTP daemons run as `nobody`.

Note that it is extremely important that the process UID be changed using the `setuid()` system call, which changes both the "real" and "effective" UID of the process. DO NOT use the `seteuid()` call (which only changes the "effective" UID of the process) because many Unix systems will allow the process to switch back to the "real" UID (root in this case) at will.

Breaking Out of `chroot()`

- Process cwd not in `chroot()` area:
  - Process didn't `chdir()` before `chroot()`
  - Attacker creates subdir to `chroot()` into
- Attacker forces process to `chdir("..")` to root directory
- Calls `chroot(".")` to get access to entire file system
- Exploit requires root privileges operate correctly

There is another well-known mechanism for escaping from `chroot()` restrictions, if you can get root access within the `chroot()` area. A complete write-up (with exploit code) of this mechanism can be found at:

http://www.unixwiz.net/techtips/mirror/chroot-break.html

The basic problem is that `chroot()` doesn't necessarily change the current working directory of the `chroot()`ed process. This means the attacker can force the process to effectively "`cd ..`" all the way up to the root of the file system. Once at the root of the file system, the attacker then calls `chroot()` on the current working directory. At this point, the attacker can now see the entire file system.

Even if the application developer is careful to change the working directory of the process to a directory someplace under the `chroot()` directory, the attacker can still make this exploit work. First, the attacker creates a subdirectory under the `chroot()`ed hierarchy (or uses an existing subdirectory that's already there). The attacker forces the process to `chroot()` to this subdirectory, leaving the current working directory of the process alone, which means the process' current working directory ends up outside of the `chroot()` restriction. Now we're back to the previous case of the attacker being able to walk backward up to the root of the real file system and call `chroot()` there.

However, this exploit requires the attacker to call `chroot()` to be successful—so if the program the attacker has compromised has given up root privileges, then this "escape hatch" doesn't work.

Beyond `chroot()`

- Kernel prevents "double `chroot()`"
- Application whitelisting with SELinux
- Virtual system instances:
  - Xen/VMware
  - Containers/Docker

Since we've come to understand the limitations of `chroot()` pretty well at this point, various efforts have been made to either "shore up" known issues with `chroot()` or provide alternative security functionality to restrict access that compromised applications would have.

For example, the `chroot()` breakout exploit on the previous slide was possible because an application is allowed to call `chroot()` multiple times during its lifecycle. Perhaps an application should only be allowed to `chroot()` one time. This is exactly one of the restrictions that kernel hooks like the grsecurity kernel for Linux (*http://www.grsecurity.org/*) enforces along with other security restrictions.

Newer operating systems are starting to implement even more granular process rights restrictions in the kernel, for example, the SELinux hooks in newer Linux kernels (or similar functionality in the grsecurity patches or AppArmor) and the Privileges functionality in Solaris 10 and later. With this sort of functionality, you can define down to the level of individual files what objects in the operating system a given process may read, write, or execute. Fine-grained control over other system calls is also included. Perhaps if you develop a restrictive enough "whitelist" privilege model for a given process, then the `chroot()` call wouldn't be required at all.

The difficulty with these fine-grained kernel controls is that they can be extremely complicated to configure. While these tools generally support a "learning" mode that helps administrators build policies based on watching the normal behavior of the application, the resulting policy files can be complex and difficult to audit.

A final alternative would be to make use of the various different "virtual machine" technologies that have become available. This would be just like deploying your application on a stand-alone server, except that the "server" in this case is actually a software virtual machine running (possibly in parallel with other virtual machines) inside of a larger host operating system. Aside from the savings in hardware, running applications in virtual machines allows you to quickly recover from a compromise by shutting down the compromised image and simply "rebooting" the original "gold standard" image that was originally deployed (assuming you kept an off-line copy).

"SCP-Only" Shell

I mentioned in the last section that one of the reasons that we don't just `chroot()` every single application is that there's some fairly significant administrative overhead in setting up the directory structure that the `chroot()` will happen in. To give you an example of this process, I'm going to show you a useful little application called the SCP-only shell that can provide a more secure file transfer platform than anonymous FTP. The SCP-only shell normally is configured to run `chroot()`ed, so we'll need to go through the steps of setting up the appropriate directory structure for the app.

What is SCP-Only Shell?

- A more secure option to anonymous FTP
- Creates captive user accounts:
  - Can only be accessed with SFTP or SCP
  - Each user is `chroot()`ed to their own directory
- Implemented as a special shell in `/etc/passwd` entries

There are still a number of sites that are using FTP to exchange files with customers, business partners, and the like. It's also still used a lot in web-hosting environments as a mechanism to allow users to publish updates to their content. The obvious problem, however, is that FTP is a cleartext protocol, so all those user credentials are susceptible to being sniffed off the wire. Plus, there's the session-hijacking threat to contend with.

The SCP-only shell is an SSH-based mechanism that gives you a lot of the features that people want out of anonymous or so-called "guest user" FTP servers, but in a more secure fashion because it's fully encrypted. The idea is that the SCP-only shell allows you to create "captive" accounts for users that only allow them to SCP or SFTP to a particular directory and which will never give them full interactive access on the server. In the typical implementation, each user is given their own password and a separate home directory for their own private use (though you could create shared accounts if you wish)—in this sense, it's much more like "guest user" access on an FTP server rather than true wide-open anonymous FTP access.

It's called the SCP-only "shell" because you literally enable this functionality by setting the `scponlyc` binary as the user's default login shell in their `/etc/passwd` entry. Of course, there's some other setup you have to do as well …

Getting It Working

- Build package from source *[Easy]*
- Create `chroot()` directory *[Example]*
- Configure user accounts *[There's a trick]*

I'll also show you some automounter tricks that you might find helpful ...

The SCP-only shell is an Open Source package that's not normally bundled with your OS. So, you'll need to go to the SourceForge site for the project (*http://sourceforge.net/projects/scponly*) and download the code. The package comes with a typical `configure` script: I recommend you enable both the `--enable-chrooted-binary` option to turn on the chroot()ing option as well as the option `--enable-winscp-compat`, which turns on support for SCP in addition to SFTP as well as adding backwards compatibility hacks for older versions of WinSCP (not strictly required anymore, but "be liberal in what you accept" applies here).

Once you've got the package compiled and installed, the hardest part of using the tool is setting up the directory structure that the `chroot()` is going to happen in. But I'm going to show you exactly how to do this so you can get a sense of the process you go through to `chroot()` a typical application.

Once you've got the basic `chroot()` directory structure set up, you'll want to create a user account and test things out. However, the basic `chroot()` setup doesn't really scale well to the case where you have thousands of these accounts (if you're a web-hosting provider, for example), so I'm going to show you some additional tricks to make this easier. More on this after we get the basic `chroot()` directory set up.

`chroot()` Dir Creation

1. Create basic directory structure
2. Copy binaries
3. Copy shared libraries
4. Make devices
5. Copy configuration files

No matter what application you're setting up a `chroot()` directory for, you still follow the same basic five steps:

1. Create the basic directory structure that the `chroot()`ed app is going to run in. Remember that from the perspective of the application, this directory is the entire operating system in miniature. So, you end up creating copies of directories like `/dev`, `/etc`, `/usr/bin`, and so on. The files that you have to populate into those directories depend heavily on what the application needs in order to function.
2. So, the next step is to copy whatever binaries the `chroot()`ed application needs to function into your new directory structure. Sometimes it's obvious what applications are needed, sometimes you have to consult the documentation or the application source code, and sometimes you need to be more resourceful. More on that as we go.
3. Once you've copied the necessary binaries, you will typically need a number of shared libraries to make those binaries operable. Unix systems have a tool called `ldd` that can tell you the shared library dependencies for a given binary, but you get to manually copy those libraries into your `chroot()` directory.
4. The application may also need copies of certain device files in order to function. For example, network-oriented applications may need a copy of `/dev/tcp` in order to emit packets. So, you get to learn how to create device files, too! Joy!
5. Finally, the app will also probably need copies of some configuration files—be they generic system configuration files from `/etc` or application-specific configuration files. Again, figuring out exactly which files are necessary can be tricky, but you can use the same techniques you would use to figure out which binaries or device files are created.

Let's work through a step-by-step example on the next several slides. The SCP-only shell is a fairly simple application to `chroot()` because it was designed with this functionality in mind. Other apps may have much more complicated dependencies, but at least this example will give you a taste of this process. By the way, the SCP-only shell comes with a script to set up the `chroot()` area for the application, but it doesn't do a very good job, in my opinion, of creating the most minimal possible set of dependencies.

Step #1: Create Directories

```
# mkdir -p /scponly/chroot
# cd /scponly/chroot
# mkdir -p dev etc lib64 usr/bin \
        usr/libexec/openssh
# chown -R root:root /scponly/chroot
# chmod -R 111 /scponly/chroot
```


First, you need to pick a directory where the `chroot()` will occur and flesh out the directory structure underneath that. The exact location of the `chroot()` directory generally doesn't matter—do whatever fits best with your site-specific disk layout policies. I often like to put `chroot()` directories in their own file system so that problems in the `chroot()` directory are less likely to impact the rest of the system (and vice versa).

Here, we're going to `chroot()` into `/scponly/chroot`. Under that directory, we're creating a number of subdirectories to emulate a typical OS layout. Under normal circumstances, you'd probably end up creating these directories one at a time as you worked through the various application dependencies during the `chroot()`ing process, but having done this before, I know what directories are required, and I'm just creating them all at once. I'll be populating these directories as we move along through the later steps.

Notice that you can and should use very restrictive file directory ownerships and permissions—much tighter than the default OS permissions on these directories. Remember that the typical `chroot()`ed application will give up root privileges as soon as it `chroot()`s. So, you want to make as much of the directory structure as possible owned by root so that it cannot be modified by the user operating the `chroot()`ed app. We're also setting very restrictive permissions on all directories. Specifically, we're making the mode 111—execute-only for all classes of users—which means you can access files and subdirectories under these directories but not write files or even get directory listings.

Obviously, we're going to need a directory where our remote user can upload and download files. We'll look at creating this directory later when we start adding user accounts.

Step #2: Binaries

```
# cd /scponly/chroot
# cp /usr/bin/scp usr/bin
# cp /usr/libexec/openssh/sftp-server \
      usr/libexec/openssh
# chown root:root usr/bin/scp \
      usr/libexec/openssh/sftp-server
# chmod 111 bin usr/bin/scp \
      usr/lib/exec/openssh/sftp-server
```


The SCP-only documentation states that we will need both the `scp` and `sftp-server` binaries in the `chroot()` directory structure. Other applications you `chroot()` may not be so clearly documented. In these cases, you could use the technique we'll be using in Step #4—there, we'll be using it to figure out which devices are required, but it also works for all sorts of other application dependencies, including required binaries.

In any event, once you've figured out what binaries you need, simply copy them from their normal locations in the file system to corresponding directories under your `chroot()` hierarchy. Again, we're setting very restrictive ownerships and permissions on these files.

Step #3: Libraries

- Run `ldd` against binaries from Step #2
- Merge shared library lists, copy into appropriate directories
- Also include `libnss_*.so.*`
- Shared libraries should be mode 555

However, the programs on most Unix-like operating systems are dynamically linked—meaning they depend upon one or more shared libraries to function properly. These shared libraries must also then be copied into our `chroot()`ed directory structure.

You can determine what shared libraries are required for a given program by running the `ldd` command on the binary:

```
# ldd /usr/libexec/openssh/sftp-server
        libresolv.so.2 => /lib64/libresolv.so.2 (0x00f80000)
        libcrypto.so.6 => /lib64/libcrypto.so.6 (0x00b10000)
        libutil.so.1 => /lib64/libutil.so.1 (0x00f9d000)
        libz.so.1 => /lib64/libz.so.1 (0x00224000)
[... and so on ...]
```

So, the trick is to run `ldd` on all of your binaries and create a single merged list of all the shared library dependencies. Then copy the relevant libraries into the appropriate directories under your `chroot()` directory. Note that in order to function, shared libraries must be installed mode 555 (read and execute for all). However, the directories the libraries live in can be mode 111.

One interesting note for Red Hat systems like our test server is that there seems to be an additional dependency on `/lib64/libnss_*.so.*`, which doesn't show up in the output of `ldd`. I'm assuming that one of the libraries output by `ldd` has its own dependency. Unfortunately, there's no "recursive" option to `ldd` to easily identify this dependency. You usually end up finding this because your `chroot()`ed app fails to work, and you'll end up doing some debugging—more on this process on the next slide.

Here's some shell code you might find useful for populating your shared libraries in a more automatic fashion. I'm doing some clever tricks with Perl to pull the shard library names from the output of `ldd` and using that output in a loop:

```
cd /scponly/chroot
for i in `ldd /usr/bin/scp /usr/libexec/openssh/sftp-server | \
                  perl -ne '($l) = m|\s((/usr)?/lib/\S+) \(0x|;
                            print "$l\n" if ($l);' | \
                  sort -u`; do
      cp $i .$i
done
cp /lib64/libnss_*.so.* lib64
chmod 555 lib64/*
```

Step #4: Devices

```
# ls -l /dev/null
crw-rw-rw- 1 root   1, 3 Jan 14 16:17 /dev/null
# cd /scponly/chroot/dev
# mknod null c 1 3
# chown root:root null
# chmod 666 null
```


Next, the administrator has to create any system devices that the program requires. The easiest way to figure out which devices are required is to run the program under `strace` (or `truss` under Solaris, or whatever system call tracing tool is appropriate for your OS) and see what device files are being opened. While the output of a program like `strace` is huge and difficult to read, what typically happens is that the daemon process attempts to open a non-existent device and aborts. Somewhere in the last couple of dozen lines of output, you'll see an `open()` call that fails with `ENOENT` (no such file or directory) and be able to read which device was being `open()`ed. Here's some sample output to show you what I'm talking about:

```
# ps -ef | grep sshd
root      2253     1  0 08:57 ?        00:00:00 /usr/sbin/sshd
# strace -f -p 2253
[... gobs of output not shown ...]
[pid 15766] open("/dev/null", O_RDWR|O_LARGEFILE) = -1 ENOENT (No
    such file or directory)
[pid 15766] write(2, "Couldn't open /dev/null: No such"..., 50) = 50
[pid 15766] exit_group(1)                 = ?
Process 15766 detached
[... lots more output ...]
```

The first step is to figure out the process ID of the master SSH daemon, then call `strace` with the `-f` flag (follow forked child processes) on that process ID (`"-p 2253"` in this case). Picking the relevant `ENOENT` out of this morass can be non-trivial, especially when using `"-f"`, so I recommend redirecting the `strace` output to a file and `grep`-ing for `ENOENT`. Generally, the last `ENOENT` is the one that caused the app to bomb out, so that's the one you want to look at (not all `ENOENT` errors are fatal).

Once you've figured out the device you need to create, it's time to fire up the `mknod` command. But how do you figure out the correct options to pass to the `mknod` program in order to create the appropriate device(s) in your

`chroot()` area? The easiest thing to do is just run `ls -l` on the actual system devices: The *device type* is the first letter of each line of output (either a "`c`" or a "`b`" for device files); and the other two numeric arguments, the *major* and *minor device numbers*, are displayed where the file size would normally be shown (device files have no data blocks associated with them, hence no file size need be reported). You can even copy the default ownerships and permissions from the output of `ls -l`.

Of course, even though we've created the `/dev/null` device that the app is dependent on, there may be other application dependencies that we haven't found yet. So, you go back and try your application again, and it will bomb out someplace else. Again, you look at the `strace` output to figure out what file or device you're missing and update your `chroot()` area appropriately. You keep doing this until the application "works." And if that sounds like a tedious, time-consuming process, it certainly can be. However, this same process will also help you figure out other missing application dependencies, including configuration files (Step #5, coming up next), binaries, and shared libraries. Of course, in the case of a missing binary, the `ENOENT` will occur on an `exec()` call rather than an `open()`.

Step #5: Config Files

```
# cd /scponly/chroot/etc
# cp /etc/group .
# grep ^root: /etc/passwd >passwd
# chown root:root *
# chmod 444 *
```


Finally, you'll need to create some "stub" files under `/etc`. For example, `passwd` and `group` files are required so that the `chroot()`ed `scp` and `sftp-server` commands can display user and group names rather than user and group numbers. In general, you don't need the complete contents of either file—though in this case, we're just copying in the entire `/etc/group` file because it's easier, but stripping most of the `/etc/passwd` entries.

By the way, *never* create an `/etc/shadow` file or put real encrypted password strings in the `chroot()`ed hierarchy since these configuration files are not used for authentication (the authentication happens long before the SCP-only shell invokes `chroot()`). Some security administrators have created "honey pot" machines with bogus password entries and then raised alarms when somebody tried to log into the system with one of the bogus username/password combinations, but frankly, there aren't enough hours in my day to pay attention to stuff like that.

Note that the configuration files in the `chroot()`ed directory should be world-readable but not writable.

Single User Setup

```
# groupadd scponly
# useradd -c 'Test User' -g scponly \
        -M -d /scponly/chroot//data \
        -s /usr/local/sbin/scponlyc testuser
# cd /scponly/chroot
# mkdir data
# chown testuser:scponly data
# chmod 750 data
# grep ^testuser: /etc/passwd >>etc/passwd
```


The only remaining chore is to actually create a user account that logs into the `chroot()` directory we just got done setting up and to create a directory under the `chroot()` area where this user can put files and take files from.

The `useradd` command on the slide creates a user called "`testuser`" whose shell (`-s`) is set to `/usr/local/sbin/scponlyc` (the trailing "`c`" means the `chroot()`ing version of the `scponly` binary). Note the slightly funky way were specifying the home directory (`-d`) for the user—the "`//`" marker in the directory name indicates the place where the `chroot()` should occur, and the remainder of the path gives the default directory to put the user into when giving them access. So, in this case, `scponlyc` will `chroot()` to `/scponly/chroot` and the user will start in the `data` subdirectory.

Of course, we also need to create that `data` directory and set appropriate ownerships and permissions so it's writable by the "`testuser`" account. It turns out that you also need to put a copy of the "`testuser`" password entry in the `passwd` file under the `chroot()` directory structure (that's what the `grep` command in the last line on the slide does).

Wait Just a Minute There!

- What if you have *thousands* of users?
- Maintaining individual `chroot()` areas would be painful!
- Use automount magic, tweaking original directory layout …

OK, how many of you realized the problem with our last example? What if you were a web-hosting provider with literally thousands of users with SCP-only access? Would you create a separate `chroot()` directory for each user? Of course, you wouldn't—that would be a nightmare to maintain and a huge waste of disk space to have all those duplicate shared libraries everywhere.

Wouldn't it be cool if we could have a single copy of the basic `chroot()` directory structure but still have separate `data` directories for each user? Turns out that there is a way to do this if your automounter kung fu is strong. Let me show you how …

The Automount Trick

In `/etc/auto.master`:

```
/scponly/mounted  /etc/auto.scponly
```

The `/etc/auto.scponly` map:

```
*       /             :/scponly/chroot   \
        /data         :/scponly/data/&
```

Yes, this craziness actually works...

In automount parlance, what we're doing in the `/etc/auto.master` file is defining an *indirect map* underneath the `/scponly/mounted` directory. We're going to change our user home directory entries in `/etc/passwd` so that they point to directories under this file system, rather than using the raw `/scponly/chroot` directory structure like we were using earlier. The definitions of what happens under `/scponly/mounted` are defined in the `/etc/auto.scponly` configuration file.

`/etc/auto.scponly` uses lots of cool automounter tricks, including *wildcarding* (that's the "`*`" at the front of the entry), *multiple mounts* (notice that we're driving two distinct mount points out of our single wildcard entry), and *bind mounts* (we use just a "`:`" prefix to indicate we're mounting local directories on a different mount point rather than file systems from some other server over the network). Yow!

What this entry means is when you attempt to access any directory ("`*`") under `/scponly/mounted`, two mounts are done (and, yes, they happen in the order specified in the entry) and pieced together into a coherent file system:

1. First, the `/scponly/chroot` directory is mounted on top of the root of the requested file system. So, if the system tries to access `/scponly/mounted/testuser`, then `/scponly/chroot` is mounted as this directory.
2. But each user needs their own private `data` directory. We're going to create a new directory structure called `/scponly/data`, which has individual user-specific data directories—`/scponly/data/testuser` and so on. Our automount map uses the special "`&`" macro, which means substitute whatever directory name was matched by our initial "`*`" wildcard. So, if `/scponly/mounted/testuser` is requested, then `/scponly/mounted/testuser/data` is actually mounted from the `/scponly/data/testuser` directory. Slick, huh?

By the way, my original thought had been to actually mount `/scponly/chroot` with the "`-ro`" ("read-only") option, to make it even harder for attackers. But unfortunately, with bind mounts like we're doing here, you're not allowed to set mount options that aren't set on the original file system. I suppose you could put the `/scponly/chroot` directory into its own file system and mount that read-only, but I'll leave this as an exercise to the reader.

Need to Tweak Directory Structure

- **`/scponly/chroot/data`** is now just a stub for automounting other directories
- Physical directories for user accounts should be **`/scponly/data/`**<*user*>
- **`/scponly/mounted`** will be created automatically

Of course, using the automount map from the previous slide means we've got to tweak up our original directory structure a bit. We need to create `/scponly/data` and relocate the `/scponly/chroot/data` directory we created for our "`testuser`" account to `/scponly/data/testuser`. It's important when we do this that we leave behind an empty `/scponly/chroot/data` directory so that the automounter has a mount point to place the user's `data` directory on top of.

We actually don't need to create the `/scponly/mounted` directory. The automounter will automatically create this directory for us as soon as we create the automount maps that I showed you on the previous slide and restart the automounter (`/etc/init.d/autofs` on your CentOS virtual machines).

New User Configuration Is a Snap

```
# useradd -c 'New User' -g scponly \
      -M -d /scponly/mounted/newuser//data \
      -s /usr/local/sbin/scponlyc newuser
# cd /scponly/data
# mkdir newuser
# chown newuser:scponly newuser
# chmod 750 newuser
# grep ^newuser: /etc/passwd \
      >>/scponly/chroot/etc/passwd
```


This new automount-based scheme makes adding new users very simple. Say we wanted to create an account called "`newuser.`" The home directory entry for this user needs to use the `/scponly/mounted/newuser//data` path (and we should go back and fix the home directory for our "`testuser`" account to use this scheme, too). Then we need to create a data directory for this user under `/scponly/data`—`/scponly/data/newuser` in this case—with appropriate permissions. Finally, we need to make sure the `passwd` entry for "`newuser`" gets populated into the shared `/scponly/chroot/etc/passwd` file. That's it! Three steps and you're done!

Summary

- SCP-only is a good substitute for anonymous FTP
- Hardest part is convincing users to use WinSCP
- The automount trick is useful when you have lots of directories to manage

You really need to be shutting off cleartext login and file transfer protocols in your enterprise, and the SCP-only shell is a good replacement for most places where you've historically been using "guest" or anonymous FTP servers. It's so easy to configure, that normally the biggest hurdle is getting your Windows users to start using WinSCP instead of their favorite FTP client. By the way, WinSCP supports regular FTP too, so once your users switch, they can continue to have a single file transfer client—that client will just happen to be WinSCP.

If you are using OpenSSH 5.x or later, there is now a built-in mechanism for setting up SFTP access in a `chroot()` environment. This can be turned on specifically for a particular group of users while allowing regular users of the system to SFTP in as normal. The setup is considerably less complex than setting up the SCP-only shell. For more details, see *http://www.thegeekstuff.com/2012/03/chroot-sftp-setup/*

I took a lot of time in this section going into the details of my little automounter hack because that kind of configuration ends up being useful in lots of situations where you have multiple `chroot()` directories to manage that are essentially identical. This comes up more often than you'd believe.

By the way, if you still think you need to set up an anonymous or "guest user" FTP server for some reason, there's an Appendix at the back of this book that goes into this in a great deal of detail. But really, don't do this to yourself—use SCP-only instead.

Lab Exercise

- Configuring the scponly shell
- **`chroot()`** and automounter games!

Exercises are in HTML files in two places: On the course USB media, and in the home directory of the SANS user in the lab virtual machine. These two locations contain identical copies of the exercise files—it just depends on whether you prefer to look at them from a web browser inside the lab VM (because you're working in full-screen mode) or from your host operating system.

1. Open your web browser
2. Select "File… Open…" from the browser menu (Ctrl-O is the usual keyboard shortcut)
3. This is Day 4, Exercise 1, so navigate to *.../Exercises/Day_4/index.html* and open this HTML file
4. Click on the hyperlink, which is the title of Exercise 1
5. Follow the instructions in the exercise

SELinux

For most people, their entire interaction with SELinux is turning it off during operating system installs. I think this is primarily due to a lack of understanding of the technology, which is not entirely surprising given the general lack of decent documentation out there. Hopefully, the information in this section will give you a clearer understanding of SELinux so you can at least make an informed choice before you turn it off. ☺

SELinux Topics

- Overview
- Common Troubleshooting Tasks
- Writing Policies from Scratch

The *Overview* section is a quick, high-level introduction to SELinux and some of the outermost administrative interfaces to the SELinux environment.

Next, we'll look at some *Common Troubleshooting Tasks* that tend to come up frequently in environments running SELinux, using some web server configuration examples.

Finally, in the *Writing Policies from Scratch* section, we'll actually work through a complete example of creating a brand-new SELinux policy for a real network service.

SELinux Overview

First, a general overview and high-level introduction to SELinux…

What Is SELinux?

Kernel-based access controls:

- MLS/MCS: Think Secret, Top Secret, ...
- RBAC: Fine-grained user privilege controls
- Type Enforcement: Process whitelisting, app isolation

Type Enforcement is the most commonly used piece

SELinux is a very fine-grained set of access controls implemented in a kernel module. It actually covers several different types of functionality:

Multi-Level Security (MLS)/Multi-Category Security (MCS)—These sorts of security controls are typically required at high-security sites that need to rigidly partition different categories of data (unclassified, classified, secret, top secret, etc.) on their networks. Outside of these kinds of environments, however, this level of access control is not widely used or needed. Luckily, you can simply ignore this functionality in a typical SELinux configuration by simply not using it—all objects will be placed in the same minimum-security level by default and can freely interact with each other.

Role-Based Access Control (RBAC)—One historic problem with the Unix security model has been "all or nothing" administrative access via the root account. RBAC is an attempt to address this deficiency by allowing sites to assign specific aspects of administrative privilege to a number of different roles. For example, you might have a "network administrator" role that allows somebody to configure the network interfaces on the device, while the "printer admin" role allows somebody control the print queues. The end game would be to completely disable the root account and just assign people to specific roles based on their job function.

However, there are a couple of issues:

- Nobody has actually published a meaningful RBAC policy for a typical enterprise Linux environment—even a decent subset of one—and creating one from scratch is an enormous effort. Even if somebody did create and publish a straw-man policy, it might end up being too site-specific to be useful to other organizations.
- While different flavors of Unix support RBAC in their kernels, every vendor's implementation is completely different from a configuration syntax and administrative implementation perspective. So, if you spend a lot of time crafting an SELinux-based RBAC policy, you'd have to spend a bunch more time "porting" that policy to your Solaris, AIX, etc. systems.

This is why `sudo` still tends to be quite popular. It gives you 80% of the fine-grained access controls you'd get from RBAC, and it's both easier to configure and portable across multiple Unix variants.

Type Enforcement (TE)—Type Enforcement is essentially an application "whitelisting" facility. Type Enforcement policies attempt to specify exactly which components of the operating system (files and directories, devices, sockets, etc.) a given application needs to interact with and what level of access the application needs (read access, write access, etc.). If an application attempts to stray outside the bounds of the defined policy, the kernel refuses to grant access to the requested resource. More often than not, this will cause the application to fail.

The goal is to prevent an application that has been compromised by an attacker from misbehaving in ways that would allow the attacker to compromise the larger systems. So, Type Enforcement is really an "application isolation" strategy, similar to `chroot()`. However, where `chroot()` can really only control the application by limiting to a particular subset of the file system, Type Enforcement can control many other aspects of the application's behavior (access to network sockets, for example, or finer-grained access controls to specific files) and still allow the application to run in the default OS directory structure (meaning no need to set up a `chroot()` directory with copies of binaries, libraries, etc.). You could even run an application with a combination of `chroot()` and Type Enforcement restrictions, though most organizations see Type Enforcement functionality as a superset of the security controls they get in a typical `chroot()` environment.

These days, most sites are at most adopting Type Enforcement only and largely ignoring both MLS/MCS and RBAC. The most popular Type Enforcement strategy is the so-called "targeted policy" implemented by default in Red Hat Enterprise Linux (and also used by Gentoo Linux). As of RHEL5, the targeted policy covers nearly all standard OS daemons that can be accessed from outside the box and a number of other programs as well, while leaving normal user and internal administrative processes alone (thus reducing the pain of implementing SELinux in most environments).

The original [SELinux] policy published at the NSA was the strict policy. Its goal was to lock down the entire operating system, controlling not only the daemons that live in system space but controlling the user space as well. Strict policy … adds the largest burden on users …

During the development of Fedora Core 2, we attempted to use strict policy as the default policy. We had multiple problems with this because the strict policy was governing the way users were running their systems. We had to cover all possible ways that a user would be able to setup their system. As you can imagine, we had a ton of problems and bug reports. Most people, when confronted with SELinux at that time, just wanted it turned off…

After our experiences with the strict policy, we went back and reflected on what our goals were. We wanted a system where the user was protected from System applications that were listening on the network.

These applications were the doors and windows where the hackers would enter the system. So we decided to target certain domains and lock them down while continuing to leave user space to run in an unconfined nature. Targeted policy was born …

From early Red Hat documentation

Alternate SELinux Universe

- *Objects* are anything in OS: Files, devices, sockets, processes
- Objects have SELinux *contexts*—just labels chosen by us
- SELinux *policy* defines how an object in one context interacts with other objects

All of this is completely independent of normal Unix ownerships and privileges …

When SELinux talks about *objects*, it just means something in the operating system—whether that's a file, a directory, a device, a network socket, a process, or what have you. Each object is assigned a label that describes the *context* associated with that object. These context labels are just a set of conventions established by the folks who developed the SELinux policy on your system. When you're developing your own policy for new applications, you also end up creating new labels to identify the pieces of the operating system that are specific to your application.

SELinux policy rules control how an application running in a particular context, such as Apache web server running in the "`httpd_t`" context, can interact with other objects in the operating system—files in the web docroot, for example, which have context type "`httpd_sys_content_t`" in the default targeted policy. A policy statement might say that processes in the `httpd_t` context may read files that have `httpd_sys_content_t` context. Anything not specifically permitted by the SELinux policy is denied, so if the policy only grants the web server read access to files in the docroot, then if the process tried to overwrite one of those files (web defacement), the kernel would block the write attempt, returning an error code to the process. Depending on how the application is coded, the process may catch the error and keep running, or terminate and give up.

It's important to understand that SELinux essentially exists as an "alternate universe" from the normal Unix ownership and permissions rules that you're used to. That being said, it's important to maintain good discipline around normal Unix ownerships and permissions, because, again, most sites only implement SELinux Type Enforcement on certain critical processes and rely on normal Unix permissions to control access for interactive user sessions.

"-Z" Shows Contexts

```
# ps -eZ | grep httpd
system_u:system_r:httpd_t:…  2773 ?  00:00:00 httpd
system_u:system_r:httpd_t:…  2775 ?  00:00:00 httpd
system_u:system_r:httpd_t:…  2776 ?  00:00:00 httpd
system_u:system_r:httpd_t:…  2777 ?  00:00:00 httpd
system_u:system_r:httpd_t:…  2778 ?  00:00:00 httpd
…
# ls -dZ /var/www/html
d… system_u:object_r:httpd_sys_content_t /var/www/html
# id -Z
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
```


On a system that includes SELinux functionality, many of the standard OS programs have an added "`-Z`" flag, which displays the SELinux context information.

The context label is actually a triple of "user", "role", and "type." But if you're not using RBAC, "user" and "role" generally don't matter. The "user" field tends to be either "`system_u`" (for system processes and other objects in the file system) or "`user_u`" (for normal users and their processes). Processes and users tend to get shoved into the role "`system_r`" by default, while static objects like files in the file system are labeled as "`object_r`" in general. It's the "type" field that's really used to discriminate between objects in the typical Type Enforcement targeted policy. Note that the `*_u`, `*_r`, and `*_t` suffixes are purely conventional—they're just put there to make it easier to interpret the labels.

You'll also often see a string like "`s0-s0:c0.c1023`" appended to each context. This is the MLS/MCS label for each object. The first part is a range of "sensitivity levels" (think *confidential*, *secret*, and so on) that can see the object—by default all objects in the OS are shipped at sensitivity level zero (the lowest level) so that they can be used by all users on the system. The "`c0.c1023`" is a range of "categories"—think "proprietary" and so on—which, when combined with the sensitivity label, lets you say things like "*confidential and proprietary*" in you MLS/MCS policy. Again, the default here is "`c0.c1023`", covering the entire range of possible values. So, once again, any object on the system can interact with any other object without interference from the MLS/MCS policy.

Labels are inherited. If the `httpd` process runs a CGI script, that script will run in the `httpd_t` context unless the SELinux policy explicitly says otherwise (this is called a *transition* and does happen, as for example when `init` starts up a new daemon that wants to run in its own private context). Similarly, if you create a new file in a directory, it will tend to inherit the context of the parent directory by default.

You'll notice that our user session as reported by "`id -Z`" is labeled with "`unconfined_t`". The targeted policy has a generic rule that doesn't put any restrictions on objects in the `unconfined_t`. Effectively these objects, which include all user sessions and the processes they spawn, will not be constrained by SELinux at all, though they are still subject to normal Unix ownership and permission restrictions.

Is It Turned On?

```
# sestatus
SELinux status:                 enabled
SELinuxfs mount:                /sys/fs/selinux
SELinux root directory:         /etc/selinux
Loaded policy name:             targeted
Current mode:                   permissive
Mode from config file:          permissive
Policy MLS status:              enabled
Policy deny_unknown status:     allowed
Max kernel policy version:      28
# setenforce 1
# getenforce
Enforcing
```


For most people, the only question they have about SELinux is, "Is that (expletive deleted) SELinux running?" SELinux can be enabled or entirely disabled, but if enabled you have the choice between *permissive* and *enforcing* mode. *Permissive* means that the kernel will log violations of the SELinux policy for the machine but still allow the application to continue. Typically, this mode is used for testing and development of new policy. If the system is in *enforcing* mode, then the kernel will terminate applications that violate policy.

`sestatus` will tell you whether or not SELinux is enabled and what mode it's running in. You can switch between permissive and enforcing mode with the `setenforce` command: Use "`setenforce 0`" (or "`setenforce Permissive`") to set permissive mode, "`setenforce 1`" (aka "`setenforce Enforcing`") turns on enforcing mode. `getenforce` will tell you which mode you're in.

Note that changes you make with `setenforce` will not persist across reboots. To make persistent changes, edit `/etc/sysconfig/selinux`, where you can choose to enable or disable SELinux and what mode to run in by default. If you're planning on changing between enabled and disabled states, you have to do it via a reboot.

SELinux Booleans

```
# getsebool -a | grep http
…
httpd_can_network_relay --> off
httpd_can_sendmail --> off
httpd_dbus_avahi --> on
httpd_enable_cgi --> on
httpd_enable_ftp_server --> off
httpd_enable_homedirs --> off
httpd_execmem --> off
…
```


When enabled, the standard targeted SELinux policy is enabled by default. However, the policy modules for many applications include optional functionality that can easily be enabled or disabled via SELinux *booleans*—basically, simply on/off switches for the various bits of functionality.

You can get a list of all possible booleans and their current settings with "`getsebool -a`", or you can specify a single boolean to query as shown in the final example above. As you might expect, `setsebool` allows you to turn a given boolean on or off (1 for on, 0 for off). Note, however, that the changes you make with `setsebool` will not persist across reboots: Use "`setsebool -P …`" to make persistent changes.

In most cases, the name of the boolean pretty well describes what functionality it enables, particularly if you're familiar with the normal functioning of the application in question. However, if you're curious about exactly what each boolean enables, your only recourse is to get the source RPM for the SELinux policy for your system and look at the source code (the actual running policy files are stored in a binary format under `/etc/selinux` and are not human readable or easily reverse-engineered). Obviously, this is not an ideal method of documentation. More information on obtaining the SELinux policy source RPMs will be provided a little bit later in this module.

In the targeted policy that was implemented in RHEL5, each application had a `*_disable_trans` boolean associated with it. The idea was that when you turn these booleans on, it was supposed to allow the application to violate the SELinux policy, even when the system as a whole was in enforcing mode. Thus, they were supposed to be a temporary escape mechanism to test new functionality or quickly work around SELinux-related problems in production. Unfortunately, in practice, the `*_disable_trans` booleans didn't really end up working. In fact, these booleans have been completely removed from recent versions of the targeted policy and you won't find them at all in RHEL6 and later.

Common Troubleshooting Tasks

Now that we've oriented ourselves a bit in this new SELinux universe, let's look at some of the kinds of problems you will run into when you start using SELinux. This will also allow us to introduce some new commands for controlling objects in an SELinux environment.

Some `httpd` Examples

Move docroot to new partition:

- *Setting proper SELinux context on new directory*

Using an alternate port number:

- *Granting access to network ports*

While the default SELinux policy works OK with the default configuration as provided by your OS vendor, the minute you start changing things, you start creating SELinux policy violations. On enforcing systems, this means your application gets terminated by the kernel. At this point, most people give up and disable SELinux because they don't know how to troubleshoot and fix the problem. So, let's work through some common kinds of failures.

The first kind of failure happens when you try to use a different directory configuration than the default—like when you try to get your web server to use an alternate document root. The normal problem here is that the alternate directories you're using don't have the proper SELinux context labels on them, so access is not permitted by the standard policy. The fix is to put the right labels on the new directories, and I'll show you how to do that.

Another common failure mode happens when you try to use an alternate port number for a service—for example, trying to get your web server to use an alternate port like 8888/tcp. Just like files and directories, sockets have contexts as well and you have to make sure the port you're planning on using is associated with the proper context. So, we'll also get to see how to manage contexts on port objects.

However, there are some problems that are not easily solved simply by relabeling. In these cases, we actually have to extend the default policy with our own rules. We'll look at these tools in the final section when we look at creating our own policies from scratch.

Docroot of Doom!

The scenario:

- You decided to create `/docroot` dir
- Updated `httpd.conf`
- Added some content

Mysterious errors when you restart Apache

Must be an SELinux problem, right?

So, suppose you decided to create a new `/docroot` directory to hold your web content. After creating the directory, you update your `httpd.conf` file appropriately and restart the server. In RHEL5, the server would bomb with a mysterious error message. Looking in your Apache error logs, you see:

```
[notice] SIGHUP received.  Attempting to restart
Syntax error on line 281 of /etc/httpd/conf/httpd.conf:
DocumentRoot must be a directory
```

In RHEL6 and later, the web server starts up, but when you go to access any document under `/docroot`, you get a `"Permission denied"` error:

```
[Wed Jun 10 16:14:54.732335 2015] [core:error] [pid 15647]
(13)Permission denied: [client ::1:35764] AH00035: access to
/index.txt denied …
```

Now you're a little crazed because you can do an "`ls -ld /docroot`" and see pretty clearly that it's a directory and the permissions are set correctly. And all the files under `/docroot` are world-readable.

You're pretty sure this problem must have something to do with that wacky SELinux thing, but how can you be sure?

Is It Really SELinux?

```
# ausearch -f /docroot
----
time->Wed Jun 10 16:14:54 2015
…
type=AVC msg=audit(1433967294.731:635): avc:  denied
{ getattr } for  pid=15647 comm="httpd"
path="/docroot/index.txt" dev="dm-0" ino=264552
scontext=system_u:system_r:httpd_t:s0
tcontext=unconfined_u:object_r:default_t:s0
tclass=file
```

Seems like it could be an SELinux issue …

SELinux uses `auditd` for logging events. That means we can use our old friend `ausearch` to find out if we have any events related to the `/docroot` directory:

```
# ausearch -f /docroot
----
time->Wed Jun 10 16:14:54 2015
…
type=AVC msg=audit(1433967294.731:635): avc:  denied  { getattr }
for  pid=15647 comm="httpd" path="/docroot/index.txt" dev="dm-0"
ino=264552 scontext=system_u:system_r:httpd_t:s0
tcontext=unconfined_u:object_r:default_t:s0 tclass=file
```

Obviously, these audit logs are not designed with readability as the foremost goal (actually if you're on the console using the GUI, these SELinux alerts will also pop up from time to time and you can use the GUI-based tools to get more detailed human-readable output). But you can puzzle out what's going on here once you've been exposed to the madness for long enough:

1. First, we see that access is being "`denied`" to "`getattr`". "`getattr`" is short for "get attributes", and corresponds to trying to `stat()` a file or directory to figure out whether it exists and/or what kind of object it is.
2. The process triggering the denial is "`httpd`" (pid `15647`).
3. It's trying to access "`/docroot/index.txt`" (inode `264552` on `/dev/dm-0`) which is a file ("`tclass=file`").
4. The context of the `httpd` process (the source context, "`scontext`") is `…:httpd_t` and the context of `/docroot` (the target context, "`tcontext`") is `…:default_t`.

Other `ausearch` options that might be useful in these cases include:

```
ausearch -c httpd      # search for entries where the "command" matches httpd
ausearch -se httpd     # search for entries where the SELinux context matches httpd
ausearch –m AVC        # search for SELinux alerts with "type=AVC"
```

`aureport –a` can also be useful to see a summary of `type=AVC` alerts. You can then use `ausearch –a` to get more details about specific events:

```
# aureport -a

AVC Report
============================================================
# date time comm subj syscall class permission obj event
============================================================
1. 05/14/2015 10:06:54 ? system_u:system_r:init_t:s0 0 (null) (null) (null)
unset 453
…
8. 06/10/2015 16:14:54 httpd system_u:system_r:httpd_t:s0 4 file getattr
unconfined_u:object_r:default_t:s0 denied 634
9. 06/10/2015 16:14:54 httpd system_u:system_r:httpd_t:s0 6 file getattr
unconfined_u:object_r:default_t:s0 denied 635
# ausearch -a 635
----
time->Wed Jun 10 16:14:54 2015
type=SYSCALL msg=audit(1433967294.731:635): arch=c000003e syscall=6
success=no exit=-13 a0=7fba85170f50 a1=7fff0efc0e40 a2=7fff0efc0e40 a3=0
items=0 ppid=15645 pid=15647 auid=4294967295 uid=48 gid=48 euid=48 suid=48
fsuid=48 egid=48 sgid=48 fsgid=48 tty=(none) ses=4294967295 comm="httpd"
exe="/usr/sbin/httpd" subj=system_u:system_r:httpd_t:s0 key=(null)
type=AVC msg=audit(1433967294.731:635): avc:  denied  { getattr } for
pid=15647 comm="httpd" path="/docroot/index.txt" dev="dm-0" ino=264552
scontext=system_u:system_r:httpd_t:s0
tcontext=unconfined_u:object_r:default_t:s0 tclass=file
```

So How Do I Fix This?

- Server works fine with the default document root (`/var/www/html`)
- Need to set the same SELinux *context* on our new document root directory
- Time for some more new commands…

We know that the SELinux policy allows Apache to operate fine with the default document root—`/var/www/html` on Red Hat systems. That implies that if we could just copy the context that's set by default on `/var/www/html` and apply it to our new document root, then things would probably start working.

We've already seen "`ls -Z`" to display the current context of a file or directory. But now we need to learn how to change context labels on file system objects…

Lab Exercise

- Troubleshooting SELinux Part I
- Try it, you might like it …

Exercises are in HTML files in two places: On the course USB media, and in the home directory of the SANS user in the lab virtual machine. These two locations contain identical copies of the exercise files—it just depends on whether you prefer to look at them from a web browser inside the lab VM (because you're working in full-screen mode) or from your host operating system.

1. Open your web browser
2. Select "File… Open…" from the browser menu (Ctrl-O is the usual keyboard shortcut)
3. This is Day 4, Exercise 2, so navigate to *.../Exercises/Day_4/index.html* and open this HTML file
4. Click on the hyperlink, which is the title of Exercise 2
5. Follow the instructions in the exercise

Setting File/Directory Contexts

Two-step process:

- First, use "`semanage fcontext ...`" to update policy
- Then "`restorecon -FR ...`" to fix contexts

Files created later automatically inherit context of parent dir

Let's start by using "`ls -Z`" to list the contexts on `/docroot` and `/var/www/html`. Actually, we already know the context on `/docroot` because it was the "target context" in the `audit.log` output, but let's see it again side-by-side:

```
# ls -Zd /var/www/html /docroot
d... root unconfined_u:object_r:default_t:s0 /docroot
d... root system_u:object_r:httpd_sys_content_t:s0 /var/www/html
```

OK, we need to set the context "`system_u:object_r:httpd_sys_content_t`" on `/docroot` and all the files underneath that directory. Just like `chown` and `chmod`, there's a `chcon` command for changing SELinux contexts on files and directories. However, the changes you make with `chcon` are not persistent. So, in general, `chcon` is not the way to go here (though it's occasionally useful for making quick changes for testing purposes).

In order to make useful changes, you have to create a new entry in SELinux's internal file context policy table. We do this with "`semanage fcontext ...`":

```
# semanage fcontext -a -t httpd_sys_content_t '/docroot(/.*)?'
# semanage fcontext -l | grep docroot
/docroot(/.*)?  all files  system_u:object_r:httpd_sys_content_t:s0
```

The first command above adds ("`-a`") a file context entry for `'/docroot(/.*?)'`—a regular expression that matches the `/docroot` directory itself and everything underneath it. In this case, we're specifying the default type attribute ("`-t`") "`httpd_sys_content_t`" for this directory and everything underneath. Since we didn't specify the user attribute or the role portions of the label, these properties were set to sensible default values. You can see this when we use "`semanage fcontext -l`" to list the file context table entries.

Note that if you make a mistake when you add a file context entry, you can delete it by using "`semanage fcontext -d …`"—you use exactly the same syntax and all the same arguments you used when you did the "`-a`" version to add the rule initially.

While "`semanage fcontext …`" establishes a default policy for a given directory, it doesn't actually change the existing context labels. The preferred method for changing the labels is the `restorecon` command, which looks at the defaults in the file context table and applies the appropriate labels to the directory you specify:

```
# restorecon -FRvv /docroot
restorecon reset /docroot context user_u:object_r:default_t:s0-
>system_u:object_r:httpd_sys_content_t:s0
restorecon reset /docroot/index.txt context user_u:object_r:default_t:s0-
>system_u:object_r:httpd_sys_content_t:s0
```

Here, we're using the recursive ("`-R`") option to make sure we adjust the context labels for `/docroot` and all our content beneath it, and the verbose ("`-vv`") option so we can see the changes as they're being made. "`-F`" forces `restorecon` to change the user and role portions of the context along with the type. We can confirm that our changes fixed the problem by attempting to access documents from our web server. You'll find that our changes fix the issues we were having.

One more useful tidbit: Suppose you've been making massive file system changes and you want to make sure that the file contexts on all directories reflect the default settings in the file context table. This often comes up after you've done a full restore of your file system, because many backup utilities may not capture the SELinux context information associated with files and directories. If you "`touch /.autorelabel`" and then reboot the system, then during the boot process, the system will do a "`restorecon -R -F`" on all local file systems. Doing this during a reboot is recommended because it will be done very early in the boot process before any of the normal OS daemons have started—changing contexts on files and directories used by a process while that process is running can yield weird behavior. Note that doing a `restorecon` on the entire file system can take several minutes, so this isn't something you want to do all the time (at least if you value your uptime numbers).

Alternate Port Number? SELinux Says, "FAIL!"

The scenario:

- You want a dev server on port 8888/tcp
- Change `httpd.conf` and restart
- SELinux terminates your process

The fix:

- Figure out context for `httpd` ports
- Add your new port to this context

Now let's look at how SELinux controls access to socket and port objects. Suppose you decided you wanted to run your web server on port 8888/tcp. After updating your `httpd.conf`, you attempt to restart the server and it gets killed due to an SELinux policy violation (again confirmed via the `audit.log` file).

As was the case with our earlier docroot example, the web server worked fine when it was bound to port 80/tcp. So, there must be some SELinux context associated with 80/tcp that allows the web server to bind to that port. Clearly, this context is not associated with 8888/tcp by default. Can we change that? Of course!

Dealing with Port Contexts

```
# semanage port -l | grep ' 80,'
http_port_t  tcp  80, 443, 488, 8008, 8009
# semanage port -a -t http_port_t -p tcp 8888
# semanage port -l | grep ' 8888,'
http_port_t  tcp  8888, 80, 443, 488, 8008, 8009
```

Cannot reduce default port list without changing and recompiling default policy!

It turns out that we also use the `semanage` command to control port contexts, and you can see from the example on the slide that things are not much different from when we were using it to manage file contexts.

First, we look for port 80 in the output of "`semanage port -l`" so we can figure out the context name we need to associate with our new port 8888/tcp. Note that I'm `grep`-ing for "<*space*>`80`<*comma*>" here so that I match only port 80 in the output, and not ports like 8880, etc.

Once we figure out that the context we want to use is `http_port_t`, we can use "`semanage port -a` ..." to add our new port to the list of ports associated with this context. Frankly, I wish the "`-p`" option took a sane argument like "8888/tcp", but I wasn't consulted when the command was being created. So, you're left with the kind of backward command syntax you see in the example above.

We can confirm our changes by using "`semanage port -l`" again. You'll notice that there are a bunch of other port numbers associated with the `http_port_t` context. If you weren't ever planning on using those ports, you might try using "`semanage port -d` ..." to remove those ports from the list, thus further tightening the policy constraints on your web server. And when you try to do that, you see the following:

```
# semanage port -d -t http_port_t -p tcp 8009
ValueError: Port tcp/8009 is defined in policy, cannot be deleted
```

Yep, that's right. Removing these default ports would actually require modifying the source policy! Frankly, this is a bit of a bug.

Lab Exercise

- Troubleshooting SELinux Part II
- How about another helping?

Exercises are in HTML files in two places: On the course USB media, and in the home directory of the SANS user in the lab virtual machine. These two locations contain identical copies of the exercise files—it just depends on whether you prefer to look at them from a web browser inside the lab VM (because you're working in full-screen mode) or from your host operating system.

1. Open your web browser
2. Select "File… Open…" from the browser menu (Ctrl-O is the usual keyboard shortcut)
3. This is Day 4, Exercise 3, so navigate to *.../Exercises/Day_4/index.html* and open this HTML file
4. Click on the hyperlink, which is the title of Exercise 3
5. Follow the instructions in the exercise

Policies from Scratch

Our last example gave you a taste of using `audit2allow` and creating some SELinux policy. But what if you weren't just extending the existing policy for an application, but instead had to create an entirely new policy for a brand-new application not currently covered by the default targeted policy. That gets a whole lot more interesting…

Prerequisites

Use the *SELinux Reference Policy:*

- Creates conventions for shared resources
- Provides macros for common policy rules
- Unfortunately, documentation is lacking

Need `selinux-policy-devel` RPM

Also want `selinux-policy` *source RPM* installed

In the bad old days, you had to create raw SELinux policy from scratch. Imagine how awful that must have been. Consider a common shared service like Syslog—not only did you have to create a policy that allows Syslog to do all the things it needs to do (but not allow it to do too much) but then you had to negotiate access to all of the locations in the OS that are shared with other applications (`/etc`, `/var/log`, and so on). Plus, every other application that talks to Syslog would need to share the same set of allow rules to make that happen. Change any single piece and your changes would have to ripple out to the rest of your policy. It's a nightmare.

So, the idea for the Reference Policy was born. The Reference Policy establishes naming and usage conventions for shared resources in the operating system. It also creates macros for common access needs—sending log messages to the Syslog daemon is an example of one such macro. The bad news is that the Reference Policy is still actively being developed, which leads to two problems:

1. It's still changing rapidly. This means that from release to release you may find that policy files need to be updated to reflect changes to the Reference Policy macros. This is particularly a problem when transitioning between major RHEL releases—years of policy development have happened from one release to another.
2. The documentation is sketchy… at best. Really, the best way currently to learn about the Reference Policy is to read the source.

So, we need to get a copy of the source. The current development site for the Reference Policy is *https://github.com/TresysTechnology/refpolicy/* and you can find the download link there. But if you're using RHEL, then you're better off getting the source for the version that Red Hat used.

Happily, Red Hat makes source RPMs available for all of the Open Source software included in its releases. You'll find the source RPMs at Red Hat's site or your favorite CentOS mirror in a location like:

http://vault.centos.org/<osvers>/updates/Source/SPackages//selinux-policy-<vers>.src.rpm

When you install the source RPM, it unpacks itself into `/root/rpmbuild/SOURCES`. You'll find the files on this path in the virtual machine I gave you at the start of class. The file that actually contains the source code you want to look at is the `serefpolicy-`*<vers>*`.tgz` tar file. There's also `serefpolicy-contrib-`*<vers>*`.tgz` that contains sample policy files you can borrow ideas from. You can unpack these files anywhere in the file system that is convenient.

In addition to the files in the source RPM for your reference material, you also must have the `selinux-policy-devel` RPM installed. This RPM includes the sample policy files, but most importantly a `Makefile` that will make the whole compilation process a lot easier.

Our Target: PortSentry

- Daemons started at boot time
- Binary in `/usr/local/sbin`
- Config files in `/etc/portsentry`
- Log files in `/var/log/portsentry`
- Binds to arbitrary network ports
- Talks to Syslog

With the prerequisites out of the way, we're actually going to create a meaningful policy file for a real network service. For our example, we're going to create a policy for the PortSentry HIDS (see the Appendix for more information on installing and using PortSentry). PortSentry is an "interesting" example because it's started at boot time like other system services, has a configuration file in `/etc`, binds to network ports, talks to Syslog, and writes its own log files in a private directory. Basically, it covers all of the different kinds of access you're likely to run into for your own applications, and best of all, there's no existing policy module for it. Yay!

The SELinux policy creation process rewards those who follow standard file layout conventions, so we're going to do that. We'll put the `PortSentry` daemon binary in `/usr/local/sbin`, tell it to look for its configuration files under `/etc/portsentry`, and to write its logs to `/var/log/portsentry`. This will minimize the impact of the new software on the existing SELinux policy and make our jobs a whole lot easier.

It also helps to understand the kinds of access your application is going to need before you start crafting policy. You won't be able to predict everything up front, but the more you can "front load" the policy creation process, the less time you'll spend iterating to your final policy. In this case, we know that PortSentry is going to need to talk to Syslog and that it's going to need access to some network ports. However, unlike most applications that will typically only bind to a single port, PortSentry's behavior is to bind to a large number of essentially arbitrary ports. This is going to make the task of crafting SELinux policy in this area a little bit abnormal for our PortSentry policy, at least as compared to the network access sections for other services.

Policy Creation Overview

1. Create working dir, skeleton policy files
2. Install initial policy module
3. Set file contexts for application files
4. Start and use application
5. Capture violations from `audit.log`
6. Update policy as appropriate
7. Repeat from Step #4 until "done"

Before we dive into creating our policy for PortSentry, let's talk about the general process of developing a new policy module. First, you need to create a working directory where you'll be developing your policy file. This directory will have a copy of that `Makefile` from the `selinux-policy` RPM that I mentioned earlier, and it's also where you'll have the various input files you'll be compiling into policy.

You typically start with a minimal policy file where you define the kinds of access you know the application will need. That basic policy gets compiled and loaded into our policy database. Once the baseline policy is loaded, you monitor your `audit.log` file for violations and then use a tool called `audit2allow` to help generate additional rules for your policy based on the `audit.log` entries. You then compile an updated version of your policy module, load it, and repeat the process. You know you have a working policy when you stop seeing alerts in the `audit.log` file.

In case it wasn't already obvious, this is normally a process you would do on a non-production system with SELinux in permissive mode. Permissive mode means that SELinux will allow the application to continue running so you see everything it's trying to do to the system and thus can generate the appropriate rules. Another reason to use a non-production system is that there are points where you're going to want to reboot the machine to both make sure the application is running in the correct context as well as to understand how the application behaves when the system is coming up.

Don't worry. Once you've figured out the policy in your test environment, it can just be installed directly on your production systems (assuming your test environment mirrors your production environment closely).

Initial Setup

```
# mkdir -p /root/selinux/portsentry
# cd /root/selinux/portsentry/
# ln -s /usr/share/selinux/devel/Makefile .
# cp ~sans/Misc/selinux/portsentry/initial-policy/* .
# ls
Makefile  portsentry.fc  portsentry.if  portsentry.te
# make
Compiling targeted portsentry module
/usr/bin/checkmodule:  loading … tmp/portsentry.tmp
/usr/bin/checkmodule:  policy configuration loaded
/usr/bin/checkmodule:  writing … tmp/portsentry.mod
Creating targeted portsentry.pp policy package
rm tmp/portsentry.mod.fc tmp/portsentry.mod
```


Your working directory can live anywhere in the file system you want. I generally keep mine under `/root/selinux`. If you're doing a lot of policy development at your site, consider using a source code control system like Subversion.

Once you have your working directory, copy the `/usr/share/selinux/devel/Makefile` into it, or make a link to this file as I'm doing in the example above. You'll also need a copy of your initial policy files. I've provided some sample policy files in your VMware image and on your course CD, so you can just start with those. We'll look at these files in a lot more detail on the next slide.

When you're ready to compile your module, just run `make`. The `Makefile` we're using takes care of running `checkmodule` and `semodule_package` for you, which is extremely convenient. The `Makefile` also runs `m4` on your policy files to expand the pre-defined macros from the Reference Policy into actual SELinux rules. If you want to see what your policy looks like after the macros have been expanded, look at the file `tmp/portsentry.tmp` in your working directory. There are a lot of comments and blank lines in this file, so if you want to just see the policy statements, you can do something like:

```
# grep -v -E '^\s*(#.*)?$' tmp/portsentry.tmp
module portsentry 1.0.0;
require {
…
# grep -v -E '^\s*(#.*)?$' tmp/portsentry.tmp | wc -l
1329
# wc -l portsentry.te
26 portsentry.te
```

As you can see in this case, there's an approximately 50x increase in the number of lines from your macro-based policy file to the final version. This is one of the reasons why it's much nicer to write your policies from the macros in the Reference Policy.

About Policy Files

`portsentry.te`:

- Contains initial straw-man policy
- Save time: Do as much as possible up front

`portsentry.if`:

- Put your own custom macros here (if any)

`portsentry.fc`:

- File context definitions go here
- What we did manually with `semanage`

SELinux policy module "packages" are generally created from three different files. The `*.te` file is the main file where you'll put your policy rules. The `*.if` file is where you're supposed to put any macros that you create specifically for your own policy. You could put these in the `*.te` file if you wanted, I guess, but it keeps things cleaner to put the macro definitions someplace else.

Finally, the `*.fc` file is used to hold file context definitions that apply to your application. These look a lot like the configuration we did earlier with "`semanage fcontext ...`". The advantage to having these context definitions compiled into your policy package is that they will be automatically loaded into the file context table when you load the policy package and you won't have to muck around with `semanage` yourself. This is one of the reasons the module package format was created in the first place.

The next several pages will cover the contents of our initial straw-man policy files and introduce you to some of the common Reference Policy macros and other syntax elements…

First, let's look at our initial `portsentry.te` file:

```
policy_module(portsentry, 1.0.0)

#### Declarations

type portsentry_t;
type portsentry_exec_t;
init_daemon_domain(portsentry_t, portsentry_exec_t)

type portsentry_etc_t;
files_config_file(portsentry_etc_t)

type portsentry_log_t;
logging_log_file(portsentry_log_t);

# You need these lines if you're using an older init-based Linux
#type portsentry_initrc_exec_t;
#init_script_file(portsentry_initrc_exec_t)

#### Policy

files_search_etc(portsentry_t)
files_search_var_log(portsentry_t)

logging_send_syslog_msg(portsentry_t)

miscfiles_read_localization(portsentry_t)
```


First, we use the `policy_module()` macro to declare our policy name and version number. Then we declare a number of context types that will be used by our application. In general, the prefix for all of the types you define should match the policy name from your `policy_module()` declaration—`portsentry_*` in our case. The extra suffixes like `*_exec_t`, `*_etc_t`, and `*_log_t` are also conventions used throughout the Reference Policy (other common types include names like `*_lib_t` , `*_tmp_t`, and `*_content_t`).

For each new type we define, there will typically be some associated macro declarations. For example, `init_daemon_domain()` is commonly used in policy files for daemons that are started at boot time. This macro defines (among other things) a *context transition* that happens whenever `systemd` or `init` runs an executable that belongs to our `portsentry_exec_t` context. The newly started process will be put into the `portsentry_t` context rather than inheriting the default context from the boot system. This transition is important so that we can write policy rules that only apply to our `portsentry_t` processes and not any other processes running on the system.

Other macros that are commonly used with our type declarations include `files_config_file()` and `logging_log_file()`, which generate the appropriate type declarations in your policy file for different types

of files. It's important to note that these declarations don't actually grant any kind of access to these types of files; they merely handle the grunt work of properly declaring the file types. We'll get to access controls in later versions of our policy.

We know our configuration files are going to live under `/etc` and our log files will be written under `/var/log`. In order to be able to even access files in these directories, we need SELinux to give us access to them. That's the purpose of the `files_search_*()` macros. `files_search_etc()` expands to an allow rule that permits read-only access to the `/etc` directory for and programs running in `portsentry_t` context. `files_search_var_log()` is actually not a pre-defined macro in the Reference Policy—we'll be creating it in our `portsentry.if`, which I'll be showing you in a moment.

Finally, we end with some macros that will be common to nearly every policy file you write. You'll use `logging_send_syslog_msg()` in the policy file for any application that needs to communicate via Syslog. Similarly, most applications need to access the various language locale files in the system, so you'll want to use `miscfiles_read_localization()`.

I mentioned we needed to define our `files_search_var_log()` macro in `portsentry.if`, so let's take a look at that file next:

```
########################################
## <summary>
##      Search the /var/log directory.
## </summary>
## <param name="domain">
##      <summary>
##      Domain allowed access.
##      </summary>
## </param>
#
interface(`files_search_var_log',`
        gen_require(`
                type var_t, var_log_t;
        ')

        allow $1 { var_t var_log_t }:dir search_dir_perms;
')
```

To be honest, I created this macro by copying another similar macro called `files_search_var_lib()` and just replacing "`lib`" with "`log`". Unless you're a whiz with `m4`, I recommend doing the same thing.

Now `gen_require()` is a macro that generates a "`require { … }`" block to declare the types we're going to be using in our policy rules. Then we have an `allow` rule with some expansions: `$1` corresponds to the argument we give to the macro (`portsentry_t` in our case), and `search_dir_perms` is a macro defined elsewhere in the Reference Policy that corresponds to a list of access permissions. After being run through `m4`, the one line `files_search_var_log(portsentry_t)` gets expanded to the following policy rules:

```
require {
    type var_t, var_log_t;
}
allow portsentry_t { var_t var_log_t }:dir { getattr search };
```

That's about the least complicated macro you're likely to run into, but it gives you a flavor of how the macros are compiled into SELinux policy statements.

Finally, we have our `portsentry.fc` file:

```
/usr/local/sbin/portsentry          --
          gen_context(system_u:object_r:portsentry_exec_t,s0)
/etc/portsentry(/.*)?
           gen_context(system_u:object_r:portsentry_etc_t,s0)
/var/log/portsentry(/.*)?
           gen_context(system_u:object_r:portsentry_log_t,s0)
# You need this line if you're using an older init-based Linux
#/etc/rc\.d/init\.d/portsentry                  --
   gen_context(system_u:object_r:portsentry_initrc_exec_t,s0)
```

In the interest of readability, I was forced to introduce line breaks in the middle of each line. In practice, the `gen_context()` macros would appear as the third column in each line above—in other words, the actual `portsentry.fc` file is only four lines long.

Starting from the left-hand side of each line, the first column looks a lot like the file specification regular expressions we were using "`semanage fcontext …`" earlier. In the rightmost column, we use `gen_context()` to define what context we want the files and directories to have. Notice that has to fully declare the user context and the role along with the type. The "`,s0`" at the end of each context label is the *security level* as used by MLS. In the default targeted policy, everything is declared at "`s0`".

The middle column in the *.fc file syntax is optional and specifies the file type. For example, the first two entries use "--" to specify that the objects being referred to are regular files. "-d" would indicate a directory, "-c" a device file and so on. In two rules, we're using wildcards to specify a directory and all files underneath that directory, so using "--" or "-d" wouldn't work here because it wouldn't cover all cases. In these situations, just leave the second column blank and your rule will apply to all object types that match the given path specifier.

OK, at this point, we've got some minimal policy description files, albeit ones that don't really grant our `portsentry_t` processes much in the way of access privileges. But it's enough to get us started. So, we type `make` in our build directory, and the `selinux-policy Makefile` automatically generates our `portsentry.pp` file for us. The next slide covers loading this policy package and some other implementation details.

Load Module, Set Contexts

```
# semodule -i portsentry.pp
# semodule -l | grep portsentry
portsentry1.0.0
# semanage fcontext -l | grep portsentry
/etc/portsentry(/.*)?                    all files
system_u:object_r:portsentry_etc_t:s0
/usr/local/sbin/portsentry               regular file
system_u:object_r:portsentry_exec_t:s0
/var/log/portsentry(/.*)?                all files
system_u:object_r:portsentry_log_t:s0
# restorecon -FR /etc/portsentry /var/log/portsentry \
    /usr/local/sbin/portsentry
```


Having generated our initial `portsentry.pp` file, we use "`semodule -i ...`" to load it. We can confirm that the install worked with "`semodule -l`". We can also confirm that the file context definitions from our `portsentry.fc` file was properly loaded using "`semanage fcontext -l`".

However, while the file context table has been updated, the labels on our files and directories have not actually been updated. We have to run `restorecon` as shown on the slide above in order to set our file contexts properly.

Ready? Set? Go!

- Make sure SELinux is in Permissive mode
- Recommend rebooting so that processes are started in proper context via `systemd`
- Start new `audit.log` file
- When system comes up, start exercising new daemons ...

Great, we've now got our initial policy loaded and our file contexts set. Again, the plan is to run the app on a system in permissive mode and capture policy violations from the `audit.log` file. We're going to use a tool called `audit2allow` to help us generate the policy rules we're missing based on the policy violations recorded in the `audit.log`.

A couple of hints before you begin:

1. Create a boot configuration and start the process as it would normally start at boot time—in other words, reboot the system to start the process. If you just run the process from the command line, it may inherit the user context and role from your interactive session rather than using the user context and role that it will have in its production deployment. This difference will be reflected in the alerts in the `audit.log` file and will make it harder when you're trying to generate policy via `audit2allow`.
2. Just before you reboot the system, tell `auditd` to start a new `audit.log` file. You can do this by sending the `auditd` process a `SIGUSR1` signal via "`pkill -USR1 auditd`". If you start with a fresh `audit.log` file each time, then you won't waste time recreating policy statements from audit.log entries that you've already seen.

After you reboot the system and your process has been started, then go ahead and use the application as normal. In the case of Portsentry, this means hitting the system with some port scans to trigger PortSentry's logging behavior (and possibly other actions). You should also exercise the application by subjecting it to at least one more reboot cycle. There may be behaviors that you get when the application is shutting down and then coming back up that you may not see at any other time.

audit2allow 2 the Rescue!

```
# grep type=AVC /var/log/audit/audit.log |
    grep portsentry | audit2allow -m portsentry

module portsentry 1.0;

require {
   type portsentry_t;
   …
}

#============= portsentry_t ==============
allow portsentry_t dhcpd_port_t:udp_socket name_bind;
…
```

SANS SEC506 | Securing Linux/Unix 63

The good news is that the SELinux packages include a little program called `audit2allow` that simplifies policy creation. You just feed your `audit.log` entries into `audit2allow` and it kicks out SELinux policy that would allow the access that's currently preventing your app from functioning. Obviously, you'll want to review the output of `audit2allow` and make sure the rules are not too permissive, but the tool can really help jump-start our policy.

When running `audit2allow`, you must specify a policy module name with `"-m"`. You can see that `audit2allow` uses this information to generate a module header at the beginning of the output. In reality, we don't care about the first part of the `audit2allow` output where the module header and "`require`" block sit. What we're really interested in are the `"allow"` rules that come after the comment—these are the rules we will need to update our policy with.

And Then the Pain …

- `audit2allow` has no clue about Reference Policy macros
- So, you get to translate manually … yay!
- Having a copy of the Reference Policy sources pays off …

Unfortunately, `audit2allow` dumps out raw SELinux policy rules. It has no clue about Reference Policy macros. You could insert the raw rules into the PortSentry policy file we're creating, but in the long term, that would end up being unmaintainable.

So, the bad news here is that you have to become an expert at translating raw SELinux policy statements back into Reference Policy macros. The good news is that there are some basic patterns that happen over and over again. So, after you've done this a few times, it gets to be pretty rote. But those first few times, you're really going to need a copy of the Reference Policy source to work with.

Don't panic! I'm going to work through this example with you to introduce you to how this process works. Stay with me …

Reference Policy Translation

Find clusters of related rules in output:

```
allow portsentry_t portsentry_etc_t:dir search;
allow portsentry_t portsentry_etc_t:file { read getattr open };
allow portsentry_t portsentry_log_t:dir { write search add_name };
allow portsentry_t portsentry_log_t:file { write read create open };
```

Two choices:

- Steal code from existing policies
- Hunt around with `find`/`grep`

The first hint is to break the process down into manageable chunks. When you look at our sample `audit2allow` output, it seems like a huge pile of confusing gibberish. But if you stare at it for a while, you'll start seeing some patterns that let you group some of the rules together in clumps like I've done below:

```
allow portsentry_t portsentry_etc_t:dir search;
allow portsentry_t portsentry_etc_t:file { read getattr open };
allow portsentry_t portsentry_log_t:dir { write search add_name };
allow portsentry_t portsentry_log_t:file { write read create open };

allow portsentry_t node_t:tcp_socket node_bind;
allow portsentry_t node_t:udp_socket node_bind;

allow portsentry_t dhcpd_port_t:udp_socket name_bind;
allow portsentry_t echo_port_t:tcp_socket name_bind;
allow portsentry_t echo_port_t:udp_socket name_bind;
...

allow portsentry_t self:capability { net_raw net_bind_service };
allow portsentry_t self:rawip_socket create;
allow portsentry_t self:tcp_socket { bind create listen };
allow portsentry_t self:udp_socket { bind create };
```

First, you see some rules related to file and directory access for objects in the `portsentry_etc_t` and `portsentry_log_t` contexts. This is where the daemon is trying to read its configuration files and write to its logs. Then there are some `node_bind` and `name_bind` rules (many of which I didn't show here in order to save space). Finally, there are some miscellaneous rules related to (raw) socket access. The trick is to focus on one chunk at a time in order to break the rule translation process up into manageable pieces.

Clearly, the `portsentry_etc_t` rules are related to the process accessing its configuration files under `/etc/portsentry`. There are lots of other daemons that have configuration files under `/etc` and many of these programs have SELinux policy files already written. So hopefully we can just steal some ideas from those pre-existing policy files.

Let's unpack the tarballs from the directory where we installed the source RPM and look around:

```
# cd /root/rpmbuild/SOURCES
# for f in serefpolicy-*.tgz; do tar zxf $f; done
# cd serefpolicy-*
# ls -d serefpolicy-*
serefpolicy-3.13.1      serefpolicy-contrib-3.13.1
serefpolicy-3.13.1.tgz  serefpolicy-contrib-3.13.1.tgz
# ls serefpolicy-contrib-*/*.te | wc -l
357
```

If you look in the `serefpolicy-contrib-`<*vers*> directory, you'll find lots of sample policy files.

Your best bet is to start with one of the simpler policy files under the services directory. I've found that `soundserver.te` is pretty useful for examples:

```
# grep etc_t serefpolicy-contrib-*/soundserver.te
type soundd_etc_t alias etc_soundd_t;
files_config_file(soundd_etc_t)
read_files_pattern(soundd_t, soundd_etc_t, soundd_etc_t)
read_lnk_files_pattern(soundd_t, soundd_etc_t, soundd_etc_t)
```

The first two lines of output are very similar to declarations we already have in our basic PortSentry policy file. But the third line could be what we're looking for. However, we need to go looking for where `read_files_pattern` is defined in the core Reference Policy so that we can understand exactly what this macro does. Time for some command-line kung fu:

```
# cd /root/rpmbuild/SOURCES/serefpolicy-<vers>
# grep -rl read_files_pattern *
…
policy/modules/kernel/files.if
policy/modules/kernel/corecommands.if
policy/modules/kernel/file system.if
policy/modules/kernel/kernel.if
policy/modules/kernel/domain.if
policy/modules/kernel/devices.if
policy/support/file_patterns.spt
```


It turns out that the first file, `policy/support/file_patterns.spt` is the file we want. Here's the declaration for `read_files_pattern`:

```
define(`read_files_pattern',`
        allow $1 $2:dir search_dir_perms;
        allow $1 $3:file read_file_perms;
')
```

This is starting to look a little bit like the "`allow`" rules in our `audit2allow` output, but `search_dir_perms` and `read_file_perms` are themselves reference policy macros that are defined in a different file. If you do another round of "`grep -rl read_file_perms`", you'll end up looking at `policy/support/obj_perm_sets.spt`. In the middle of this file, you'll find:

```
#
# Directory (dir)
#
define(`getattr_dir_perms',`{ getattr }')
define(`setattr_dir_perms',`{ setattr }')
define(`search_dir_perms',`{ getattr search open }')
…

#
# Regular file (file)
#
define(`getattr_file_perms',`{ getattr }')
define(`setattr_file_perms',`{ setattr }')
define(`read_file_perms',`{ getattr open read lock ioctl }')
…
```

Great! We seem to have found the macros we're searching for, and many more. Putting the macros from the two files together, it looks like `read_files_pattern(portsentry_t, portsentry_etc_t, portsentry_etc_t)` would expand to two rules that look like this:

```
allow portsentry_t portsentry_etc_t:dir { getattr search open };
allow portsentry_t portsentry_etc_t:file { getattr open read … };
```

This seems to be a little bit more access than the rules we dumped out via `audit2allow` are saying we need. On the other hand, using the standard Reference Policy macros is a good idea for maintainability, so let's go with `read_files_pattern` here.

If we try and match up our `portsentry_log_t` rules from `audit2allow` against the macros in `obj_perm_sets.spt`, it looks like we need `add_entry_dir_perms` (the smallest permission set that includes `write` and `add_name` capabilities) plus `read_file_perms` and `write_file_perms`. `rw_file_perms` is close but doesn't include `create`. `manage_file_perms` is a little more access than we need. Unfortunately, there's no macro in `file_patterns.spt` that comes close to matching what we need, so instead, I'm going to make a new macro in `portsentry.if` :

```
interface(`rwcreate_files_pattern',`
        allow $1 $2:dir search_dir_perms;
        allow $1 $3:file read_file_perms;
        allow $1 $3:file write_file_perms;
')
```

Note that I have to use "`interface(…)`" to define the macro in `portsentry.if` rather than "`define(…)`" as the Reference Policy does in `file_patterns.spt`. This is due to the different ways in which the two files are compiled into policy.

Now we can update our `portsentry.te` file with the following macros:

```
read_files_pattern(portsentry_t, portsentry_etc_t, portsentry_etc_t)
rwcreate_files_pattern(portsentry_t, portsentry_log_t,
portsentry_log_t)
```

So much for the file access rules; now let's tackle all those `name_bind` and `node_bind` entries. There are a lot of these, but they're all generally the same, just for different ports. Let's try searching through our policy source directory for `name_bind` and see what turns up:

```
# grep -rl name_bind *
policy/flask/access_vectors
policy/modules/kernel/corenetwork.if.in
policy/modules/kernel/corenetwork.te.in
policy/modules/kernel/corenetwork.if.m4
policy/mls
```

Those …`/kernel/corenetwork.if.*` files look promising. If you scan through those files, you find lots of macros referring to `name_bind` and `node_bind`. But remember, we need to allow PortSentry to bind to pretty much any port it wants to, so the macros like `corenet_tcp_bind_all_ports()` seem most like what we need. Since we apparently need both `name_bind` and `node_bind` access to both TCP and UDP ports, the following collection of macros seems most apropos:

```
corenet_tcp_bind_all_ports(portsentry_t)
corenet_tcp_bind_all_nodes(portsentry_t)
corenet_udp_bind_all_ports(portsentry_t)
corenet_udp_bind_all_nodes(portsentry_t)
```

The only lines we haven't dealt with from the `audit2allow` output are the following:

```
allow portsentry_t self:capability { net_raw net_bind_service };
allow portsentry_t self:rawip_socket { read create };
allow portsentry_t self:tcp_socket { bind create listen };
allow portsentry_t self:udp_socket { bind create listen };
```

Turns out, there aren't really any macros in the Reference Policy to help us with these. There are certainly socket-related macros that are supersets of the "`{ bind create listen }`" and "`{ read create }`" access lists, but since we're allowing PortSentry to bind to any port on the system, we're probably better off only allowing the minimum access to these ports that we can. So, we're just going to shove these lines into our policy file verbatim.

Thus, our "final" `portsentry.te` file looks like this:

```
policy_module(portsentry, 1.0.1)

#### Declarations

type portsentry_t;
type portsentry_exec_t;
init_daemon_domain(portsentry_t, portsentry_exec_t)

type portsentry_etc_t;
files_config_file(portsentry_etc_t)

type portsentry_log_t;
logging_log_file(portsentry_log_t);

# You need these lines if you're using an older init-based Linux
#type portsentry_initrc_exec_t;
#init_script_file(portsentry_initrc_exec_t)

#### Policy

files_search_etc(portsentry_t)
files_search_var_log(portsentry_t)

read_files_pattern(portsentry_t, portsentry_etc_t, portsentry_etc_t)
rwcreate_files_pattern(portsentry_t, portsentry_log_t, portsentry_log_t)

corenet_tcp_bind_all_ports(portsentry_t)
corenet_tcp_bind_all_nodes(portsentry_t)
corenet_udp_bind_all_ports(portsentry_t)
corenet_udp_bind_all_nodes(portsentry_t)

allow portsentry_t self:capability { net_raw net_bind_service };
allow portsentry_t self:rawip_socket { read create };
allow portsentry_t self:tcp_socket { bind create listen };
allow portsentry_t self:udp_socket { bind create listen };

logging_send_syslog_msg(portsentry_t)

miscfiles_read_localization(portsentry_t)
```

New Rules … Now What?

- Update your `*.te` file (change policy version number!)
- Build and load new policy
- Start new `audit.log`, reboot system, exercise daemon, check `audit.log`...
- Keep going until `audit.log` warnings are gone

Awesome! We've got a new `portsentry.te` file to try out, this one labeled with version marker 1.0.1. Having saved the changes to the file in our build directory, we run `make` again and install the new `portsentry.pp` file. "`semodule -l`" should show that the installed module version is now 1.0.1.

At this point, it's really a question of "lather, rinse, repeat." Start a new `audit.log` file with "`pkill -USR1 auditd`" and reboot the machine. Fire some more port scans at the box to let PortSentry do its thing. Monitor your `audit.log` for any new warnings (I think our new policy file is complete and you won't hit any additional warnings, but you never know). If any warnings show up, follow the same procedure we just went through to find the appropriate macros to update your policy file, and so on.

And Finally …

- Let the app run for a while to catch all possible behaviors
- Try testing in Enforcing mode for a while, too …
- Eventually you arrive at a complete policy
- Now push same policy out across multiple systems

In general, it's also probably a good idea to let the application run for a while, just in case there are occasional behaviors of the application—weekly cleanup tasks, for example—that you didn't catch during your testing. It's also a good idea at some point during this "burn in" period to put your test system into enforcing mode as a final check that your policy is working as expected.

Eventually, you'll have high confidence that your policy is correct and complete. At this point, you can push the policy out across your enterprise. Assuming your systems are running the same OS version, you should just be able to copy the `*.pp` file out to each machine and load it with "`semodule -i`". Don't forget to do a `restorecon` to set the right contexts on the files and directories associated with the application.

If you have different OS versions running around, you'll need to recompile a `*.pp` file for each different version you're running—but again beware of changes to the Reference Policy macros between OS revisions.

Summary

- Don't just blindly turn SELinux off
  - Security benefits
  - Helps you understand applications
- Pointers to additional docs in notes ...

Is enabling SELinux the right choice for your environment? All too often, sites don't make an informed choice on this issue—they just disable SELinux immediately because they don't understand it. Hopefully, the material in this course will clear up some of the fear, uncertainty, and doubt around SELinux and let you make an honest, intelligent decision for your organization. Maybe I've even sparked your interest in SELinux a bit.

SELinux does have the promise of providing much higher levels of security. But we're only going to work out the kinks in the default policy if people actually start using it. Creating a policy for an application also teaches you a lot about how that application interacts with the operating system (similar to having to build a directory for an app to `chroot()` into, really).

More documentation on SELinux:

A good intro to SELinux: *http://www.billauer.co.il/selinux-policy-module-howto.html*

Excellent How-To from the CentOS project: *http://wiki.centos.org/HowTos/SELinux*

The Fedora Wiki: *https://fedoraproject.org/wiki/SELinux*

Reference Policy development site: *https://github.com/TresysTechnology/refpolicy/wiki*

Lab Exercise

- SELinux capstone exercise
- Earn your SELinux merit badge!

Exercises are in HTML files in two places: On the course USB media, and in the home directory of the SANS user in the lab virtual machine. These two locations contain identical copies of the exercise files—it just depends on whether you prefer to look at them from a web browser inside the lab VM (because you're working in full-screen mode) or from your host operating system.

1. Open your web browser
2. Select "File… Open…" from the browser menu (Ctrl-O is the usual keyboard shortcut)
3. This is Day 4, Exercise 4, so navigate to *.../Exercises/Day_4/index.html* and open this HTML file
4. Click on the hyperlink, which is the title of Exercise 4
5. Follow the instructions in the exercise

DNS and BIND

Some of the material in this section was excerpted from a larger tutorial on administering DNS and Sendmail. If you are interested in more detail on DNS and Sendmail administration (as opposed to security) issues, you can download the PDF for the full course book for this tutorial from:

http://www.deer-run.com/~hal/dns-sendmail/

The *DNS and BIND* and *Sendmail* books published by O'Reilly and Associates also make useful references. There is also the BIND *Administrators Reference Manual* published by the ISC:

https://www.isc.org/downloads/bind/doc/

Another good source for DNS security configuration examples is the "Secure BIND Template" document available here:

http://www.cymru.com/Documents/secure-bind-template.html

DNS/BIND Topics

- Introduction to DNS/BIND
- Split-horizon ("split-brain") DNS
- Configuring BIND
- DNSSEC

The *Introduction* is a quick introduction to the Domain Name Service and BIND, plus an overview of common vulnerabilities in past and present DNS and BIND implementations.

Split-Horizon DNS discusses the theory behind presenting one version of your DNS information to the outside world and a completely different view internally—why and when this is useful and some architectural issues with such a configuration.

Configuring BIND presents specific configuration examples of the DNS architecture introduced in the *Split-Horizon DNS* section, and introduces many of the new security features in recent versions of BIND.

At this point, you'll have the opportunity to get some hands-on experience setting up BIND to run without superuser privileges and in a `chroot()`ed environment.

We'll finish up the module with a section on *DNSSEC*, which is the only real defense against some of the DNS security vulnerabilities we'll be discussing in the *Introduction*.

Introduction

The *Introduction* is a quick introduction to the Domain Name Service and BIND, plus an overview of common vulnerabilities in past and present DNS and BIND implementations.

What Are DNS and BIND?

- The Domain Name Service maps host names to IP addresses and vice versa
- BIND is the reference standard DNS implementation, maintained by the ISC
- Global database
  - Distributed
  - Hierarchical

The Domain Name Service (DNS) is the mechanism that internet hosts use to determine the IP address that corresponds to a given hostname. For example, if your web browser wishes to reach the home page for the SANS Institute, it must first determine the IP address for `www.sans.org`. This IP address is then used as the destination address in the packets your client sends to communicate with the remote server. Conversely, the web server at `www.sans.org` will receive the IP address of your machine as the source of these packets and will most likely attempt to determine the hostname that corresponds to this IP address. Again, DNS is the mechanism that `www.sans.org` will use to make this determination.

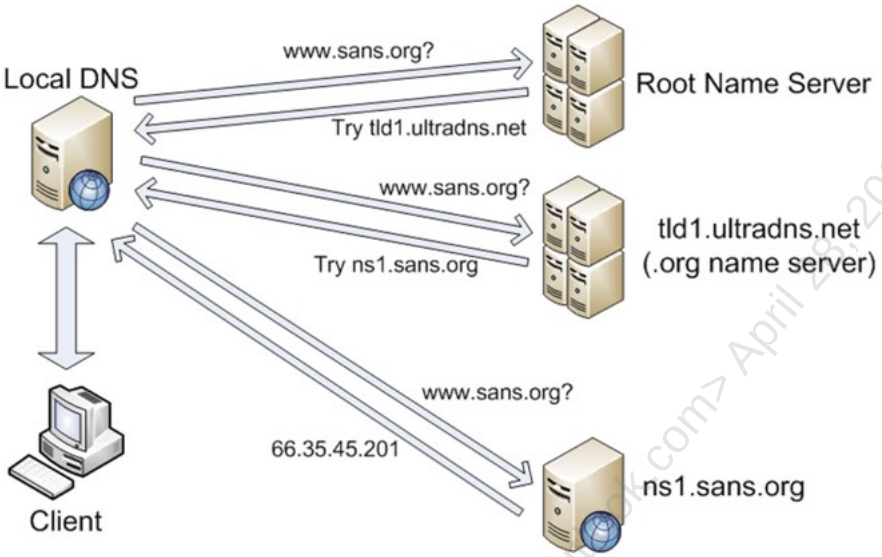
DNS can contain more information than just hostnames and IP addresses—for example, DNS can store information about the type of machine and OS platform (HINFO records), general "free-form" information about machines (TXT records), and many other types of data. In particular, DNS usually provides both internal and external machines with information about how email is to be routed to a given organization (MX records).

BIND stands for Berkeley Internet Name Daemon. From its creation, BIND has been the reference implementation for DNS on the internet and is the basis for the DNS implementation provided with most modern operating systems. For much of the 80s and 90s, BIND was developed and maintained by Paul Vixie. When Paul co-founded the Internet Software Consortium, BIND was one of the first applications to be supported by this new group. You can download the BIND sources for free from:

`https://www.isc.org/bind/`

DNS is fundamentally a distributed database system—each organization maintains its own local information. These distributed collections of information are linked in a hierarchical fashion, which is more easily demonstrated pictorially… *(see next slide)*

How It Works



Let us suppose that your local client wishes to learn the IP address of `www.sans.org`. Your client contacts a local name server that has been configured on the local client by the administrator (either statically or via DHCP, etc.). Your local DNS server actually does all of the work required to resolve the IP address and then will hand the result back to the client.

The local name server first attempts to contact one of the several *root name servers* that have been deployed on the internet and asks the question, "What is the IP address of `www.sans.org`?" The root name servers don't know the answer, but they give you a *referral* to another name server that has more information—in this case, they hand back a list of host names and IP addresses for the name servers that handle the `.org` top-level domain. Your local name server then asks one of the `.org` servers for the IP address of `www.sans.org` and receives *another* referral—this time to the list of name servers for `sans.org`. Finally, your local name server asks one of the `sans.org` servers for the IP address of `www.sans.org` and actually gets the answer back. Your local DNS server will *cache* this information for some period of time (so that other machines at your site that ask for the same information don't put extra load on the Internet DNS infrastructure) and also hand the result back to your client.

Note that in order to be able to contact a root name server to start this process, your local name server must be statically configured with the names and IP addresses of the available root name servers. This information is maintained by the InterNIC and downloaded by the administrator into a static file on the local name server.

Common Security Issues

- Giving away too much information
- DoS/Amplification Attacks
- Buffer overflows
- Cache Poisoning

Since DNS is a public networked database, one significant risk is that you give away too much information—information that can be used by people who wish to attack your systems and networks. Specific examples are given in the upcoming slides.

Because DNS is so critical for the operation of individual networks and the internet as a whole, it's also a popular target for denial-of-service-type attacks. Aside from DoS danger to your own infrastructure, badly configured DNS servers also provide attackers with an opportunity to "amplify" their attacks against other targets.

BIND has historically had buffer overflow problems in various releases. Some have led to root compromise attacks; others have simply been denial-of-service-type attacks. The best defense against these attacks is to stay up to date on the version of BIND you are running, though the *Running a Name Server* section suggests how to configure BIND to run in a `chroot`()ed environment, which can help protect you in the event of an exploitable buffer overflow.

Cache poisoning occurs when a name server has been tricked into believing erroneous information from some external source. Sometimes, this occurs by accident, but most often it is used by attackers who wish to embarrass an organization or exploit trust relationships based on hostname/address information. More on this shortly.

Too Much Info –*Public/Private*

Other organizations don't need to know that much about you:

- Addresses of your public servers
- How to route email to you

Other info is useful to your internal users—*and attackers!*

Your DNS database contains information about all of the machines in your organization. In particular, machine names and other information (HINFO and TXT records) may help an attacker locate machines that are most critical to the functioning of your organization or can be easily targeted for attack—for example, a machine called `proxy.yourdomain.com` might be interesting to an attacker wishing to penetrate the interior of your network or anonymize their attacks on other internet hosts.

Generally, the outside world needs to know very little information about your network. At a minimum, you need to advertise hostnames and IP addresses of a limited set of "public" servers: Name servers, email servers, web and FTP servers, etc. In a perfect world, an organization would be able to present one set of information to the outside and reserve a full, rich set of information for internal consumption—this is the theory behind *split-horizon DNS,* which will be covered in an upcoming section.

Too Much Info: *BIND Version*

```
% dig @ns1.sans.org version.bind txt chaos

; <<>> DiG 9.14.0 <<>> @ns1.sans.org version.bind txt …
; (1 server found)
;; res options: init recurs defnam dnsrch
;; got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 34912
;; flags: qr aa rd ra; Ques: 1, Ans: 1, Auth: 0, Addit: 0
;; QUESTIONS:
;;      version.bind, type = TXT, class = CHAOS

;; ANSWERS:
VERSION.BIND.    0       TXT      "9.14.0"
[…]
```


It is possible to query a running name server and retrieve the embedded version number string from the remote machine. Since there are known vulnerabilities in most older releases of BIND, this information can help an attacker target your machines. As we will see, administrators are able to easily change the version number string to fool attackers.

Too Much Info: *Zone Transfer*

Backup name servers transfer DNS info from master servers

Attackers will do the same to gather info about your network

Block Unauthorized Zone Transfers!

Generally, each organization runs one master DNS server and one or more slave servers for redundancy. Periodically, the slaves must contact the master and download any updates to the local DNS database—this is referred to as a *zone transfer*. By default, name servers running BIND allow any remote system to perform a zone transfer—whether that system is a legitimate name server for that domain or not. Zone transfers can even be requested from the slave name servers for your domains.

Attackers often attempt zone transfers in order to gather information about your local network. If they succeed, then they have instantly gotten all of the information about your internal hosts and networks with very little effort. Of course, a split-horizon DNS configuration can limit the amount of information an attacker will receive, but it is still a good idea to prevent unauthorized hosts from downloading your zone databases. In a later section, we will see how to configure BIND to restrict zone transfers, but it is also a good idea to block this activity at your firewall as well.

A useful resource for teaching people about the dangers of open zone transfers is DigiNinja's ZoneTransfer.me project: *http://www.digininja.org/projects/zonetransferme.php*

Licensed To: James Byrd <jeffreybubrow@outlook.com> April 28, 2020

DoS and Amplification Attacks

DNS is a popular DoS target

Misconfigured DNS servers used to "amplify" DoS effects:

- Arbitrary recursive queries
- Open cache queries

"Amplifiers" often end up DoS-ed as well

Attackers have learned that DNS is often a weak link in an organization's infrastructure. For example, an organization may spend millions on a worldwide, redundant web services infrastructure. But if the attacker can knock all of that organization's DNS servers off-line, then nobody will be able to resolve the IP addresses of those web servers, and they'll be effectively knocked off the internet. So, apply the same care to building out your DNS infrastructure and use the same DoS prevention techniques you would use with your web services.

What's become particularly prevalent lately is attackers leveraging poorly configured DNS servers run by various organizations on the internet to "amplify" their DoS attacks:

- Normally, a name server receives a request from an external name server, responds with the best information it has and does no further work—this is a *non-recursive* (sometimes referred to as an *iterative*) *query*. However, client resolvers (and sometimes other name servers as we'll see in a moment) generally do *recursive queries* when communicating with their local name server—that is, they rely on their local name server to do all the work required to look up a given piece of information (including contacting root name servers and name servers at other organizations).

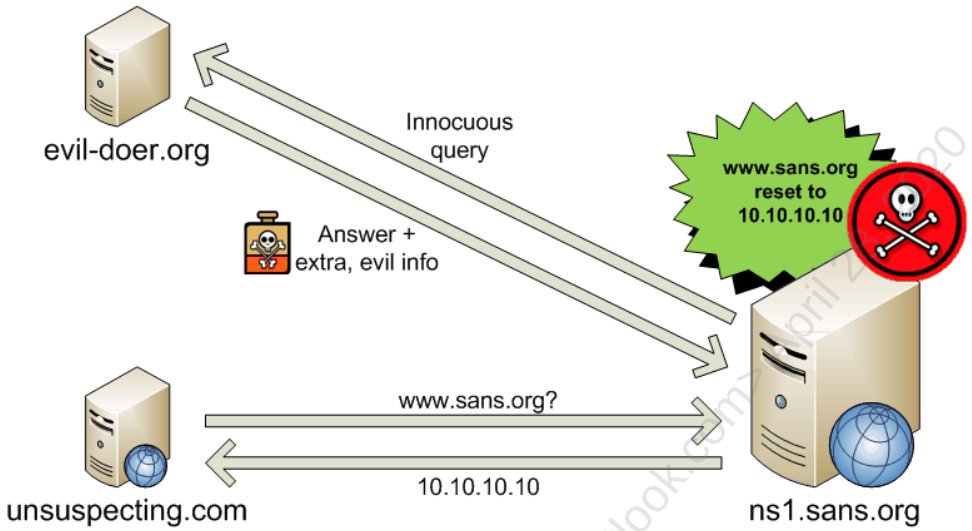
  Generally, only machines that you own should be making recursive queries via your name server. If you allow outsiders to do recursive queries via your name servers, you make it possible for attackers to force your name server to make queries on their behalf. The attacker then forces your name server to query the name server they're trying to bring down, adding your name server's bandwidth and resources to the DoS attack on the target, and incidentally making your name servers appear as a source of malicious traffic that should be blacklisted.

  Also allowing anybody to make recursive queries at your name servers makes it easier for attackers to get your name server to query the attacker's maliciously configured name servers. This can assist attackers in performing buffer overflow attacks and cache poisoning attacks against your name server.

- Up until BIND 9.4.1, the default behavior for BIND was that anybody could query your name servers for information in their cache. This was commonly used by attackers to amplify DoS attacks. The attacker would spoof a DNS query using their target as the source IP address of the bogus query. The query itself is typically a request for the list of root name servers—a very small query that results in a relatively large response. Thus, the attacker can reflect thousands of these small queries through various open servers and have a very large impact on the target machine spoofed in the source.

Aside from being a nuisance to the rest of the internet, if your name server is used as an amplifier, you may find that the bogus queries from the attacker are coming in so fast that your own name server may be DoS-ed or at least significantly degraded. You may also find yourself being blacklisted by various organizations and unable to communicate with large chunks of the internet. So, I'll be showing you some useful configuration settings to block this sort of malicious activity.

Cache Poisoning: Kashpureff



The classic cache poisoning attack described on this slide is usually referred to "Kashpureff-style" cache poisoning after Eugene Kashpureff, who used this attack in 1997 to poison the root name servers in order to advertise his alternate "AlterNIC" top-level domains. The underlying vulnerability in BIND was fixed soon after this attack and so this form of cache poisoning is no longer effective.

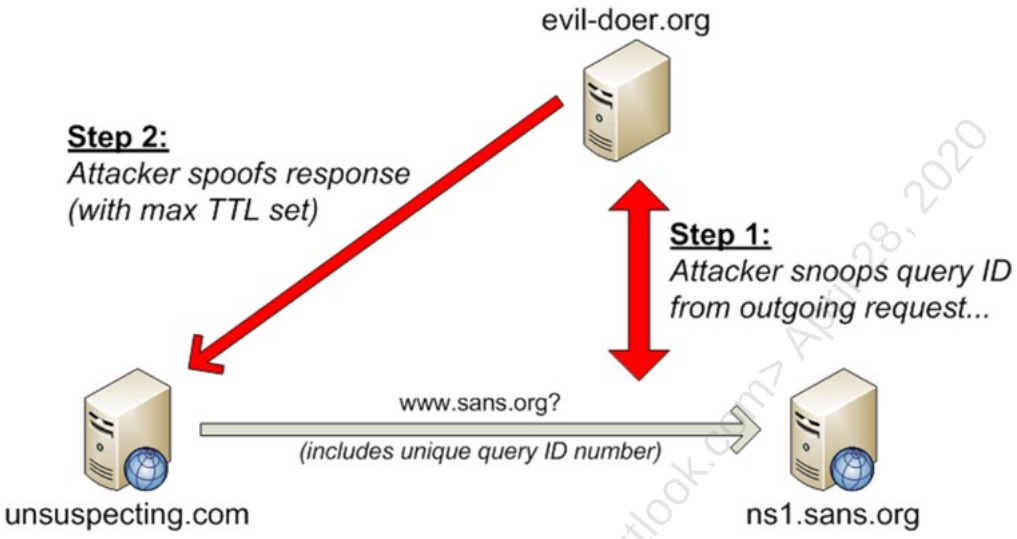
The attack is triggered when a vulnerable name server makes a query against some evil name server owned by the attacker. The attacker often triggers this initial query externally by connecting to some service on the vulnerable name server (or one of the hosts that use the vulnerable name server as their local name server)—this can cause an IP address to hostname lookup against the evil name server.

The vulnerable name server makes an innocent query against the evil name server and the evil name server hands back the proper response *plus* some extra information. Older versions of BIND would put this extra information in their cache—potentially overwriting information that they had learned from their own local zone databases.

In the example above, the evil name server has poisoned the cache on `ns1.sans.org`, convincing this machine that the IP address of `www.sans.org` is `10.10.10.10` (an invalid IP address). When any other machine queries `ns1.sans.org` for the IP address of `www.sans.org`, they will get the invalid address and be unable to reach the real web server for sans.org. Imagine, however, that the attacker had used an address that corresponded to a hard-core pornographic site (embarrassment) or an IP address that corresponded to a website owned by the attacker (phishing or man-in-the-middle attack).

Note: *The idea for the design of this slide comes from Judy Novak (thanks Judy!)*

Cache Poisoning: Spoofing



The next generation of cache poisoning attacks was really just a simple spoofing attack. DNS responses are essentially unauthenticated, so if an attacker can create and send a legitimate-looking response before the "real" name server that was queried can respond, then the host doing the lookup will happily accept the forged response containing bogus information.

The only problem from the attacker's perspective is that each DNS query goes out with a unique query ID number—the attacker's bogus response must include this ID. So, the attacker must typically first sniff the outgoing request and then incorporate the stolen query ID number into the bogus response. Obviously, the attacker must also forge the source IP address of the bogus response to make it appear that the packet comes from the remote name server that was queried. The attacker will also set the "time to live" (TTL) value in the response to be the maximum possible value so that the name server receiving the spoofed information will cache it for as long as possible.

Early releases of BIND used very predictable query ID numbers—making it possible for attackers to spoof responses without first sniffing the outgoing query. More recent releases of BIND have better randomization algorithms, though problems with these algorithms have been disclosed as recently as June 2007 (so stay up-to-date!).

Zodiac is a tool that actually implements this sort of DNS spoofing. For more information on Zodiac, see: *http://www.darknet.org.uk/2008/07/zodiac-dns-protocol-monitoring-and-spoofing-tool/*

Better Ideas for Better Exploits

- Attacker can force DNS servers to make predictable queries
- Attacker can force stream of queries
- Sends multiple responses, guessing query ID more likely
- What if attacker spoofs *glue records* instead of answers? *[Kaminsky, 2008]*

In the decade from 1998-2008, attackers continued to advance the state of the art in DNS attacks in several ways (Dan Kaminsky does a nice job of summarizing all of this research in his Black Hat slides from the 2008 conference, *http://www.slideshare.net/dakami/dmk-bo2-k8*):

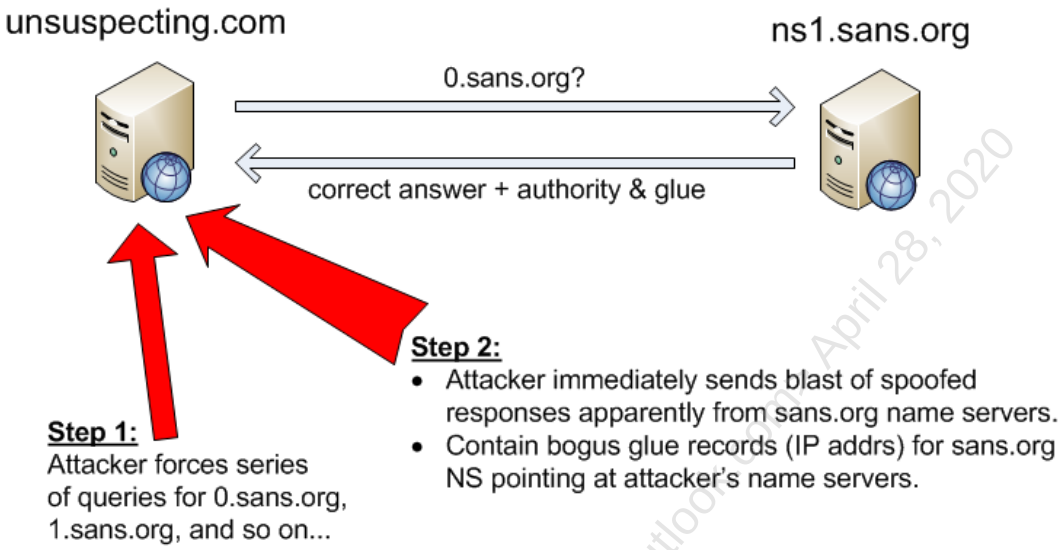
1. Attackers can often cause remote name servers to make queries for information specified by the attacker, whether because the remote site is running open name servers (more on this later) or because the attacker can force another server (mail server, web server, etc.) or client (via XSS or similar mechanism) at the remote site to make a query. This means the attacker doesn't have to wait passively for queries to spoof and it means the attacker has a very good idea of which internet name servers are going to be involved in answering the query.
2. If the attacker can force a remote name server to make one query, they can usually cause that server to make multiple queries back-to-back. This is typically done by having the remote server query a series of names like `0.sans.org`, `1.sans.org`, `2.sans.org`, etc. (you need to keep changing the hostname because the remote DNS server will cache the results of each individual query, whether positive or negative). This means the attacker can create a constant stream of queries to try and spoof, just in case their first attempt fails.
3. The spoofing attack described on the previous slide relies on sniffing the query ID out of the outgoing query—if the attacker were just guessing, they'd only have a 1-in-64K chance of hitting the right query ID. But what if the attacker sent 64K spoofed responses, each with a different query ID? Then they would be guaranteed to get a hit, right? Of course, they probably don't have the time or the bandwidth to generate that much traffic, but they can improve their odds by sending lots of spoofed responses with random query IDs.
4. But in 2008, Kaminsky had an even bigger brainwave. Every DNS response includes an "authority" section listing the NS records for the domain. Included with those NS records are so-called "glue records"—the IP addresses of the NS records in the authority section so that you can actually query them.

> Dan realized that if your spoofed response included bogus glue records that used the IP addresses of the attacker's evil name server, then when the spoofing attack worked, *all future DNS queries* that the poisoned host makes for the domain would come to the attacker's name server. The attacker effectively "owns" the entire domain from the perspective of the affected name server.

In fact, as Kaminsky points out in his presentation, the attacker could even try to do this one level up the DNS hierarchy and try to override a site's list of name servers for a top-level domain like `.com`! In other words, all of your sites queries to *any domain in* `.com` would end up going to the attacker's DNS servers. The potential for mischief here cannot be overstated because it potentially enables attackers to impersonate critical infrastructure like the Microsoft patch servers, certificate authority root servers, banking websites, mail servers, etc. *all over the Internet*. This is a huge security issue.

What's fascinating to me about the Kaminsky attack is that it took somebody *30 years* (if you count from the time that DNS first appeared on the internet) to realize that this attack was possible. You have to believe that there are plenty of other such critical exploits out there that we simply haven't thought of yet. Scary.

Cache Poisoning: Kaminsky



Kaminsky developed his own tool called "DNSRake" for performing the exploit he described in his Blackhat talk. The ideas from the talk were eagerly latched onto by other "security researchers" and further refined and weaponized. There is a Metasploit module that implements a Kaminsky-style attack.

The basic concept of the attack is straightforward:

1. Figure out a way to get the remote name server to query for a name like `0.sans.org`
2. Immediately send a large number of spoofed responses using various random query IDs that purport to come from the `sans.org` name servers (there will typically be multiple DNS servers for any given domain and the attacker can't be sure which one the remote name server is going to choose, so they have to send spoofed responses from all of them) with the bogus glue records for the `sans.org` name servers which point to the attacker's name servers.
3. Query the remote name server for what it thinks are the name servers for `sans.org` ("`dig @ns.unsuspecting.com sans.org ns`"). If the attacker sees their spoofed name servers in the reply, then the attack was successful. If the remote name server is behind a firewall and cannot by queried by the attacker, then the attacker just monitors the debugging output from their name servers and looks to see if they start getting lots of query traffic from `unsuspecting.com` for hosts in `sans.org`.
4. If the initial attack doesn't succeed, the attacker just starts over at Step #1 with a new hostname like `1.sans.org` (repeat as necessary until success).

Typically, it takes an attacker less than 10 minutes to poison a vulnerable name server.

A good write-up of the Kaminsky attack can be found at
http://www.unixwiz.net/techtips/iguide-kaminsky-dns-vuln.html

The "Fix" Doesn't Solve the Problem

"Fix" is to randomize source ports:

- Adds additional 16 bits of entropy (plus 16-bit query ID)
- Doesn't fix problem; it just makes attackers' job hard enough to prevent easy exploits

Real fix is DNSSEC, but rollout has been complicated...

The ISC and other DNS software providers have "fixed" the problem by using random source ports for outgoing queries. This means that in addition to having to guess a random 16-bit query ID for their spoofed response, the attacker also has to guess a random 16-bit destination port. 32 total bits of randomness is deemed sufficiently difficult so as to make the attack impractical—if it takes about 10 minutes to perform the attack without random source ports, then you'd expect it to take about 16 months to perform the attack against 32 bits of randomness. Assuming you have an IDS or some other way to detect the attack in progress, you should be able to shut it down before the attacker succeeds.

What's disturbing, however, is the idea that somebody might modify the Zodiac tool to sniff the outgoing query ID and source port and then fire in a spoofed referral at the name server that originated the query. In fact, such an attack would also know which of the top-level name servers was being queried, so the attacker would know which of the servers the spoofed response should appear to originate from. It requires the attacker to have the network access to snoop the outgoing queries, but it would be a very nasty exploit.

The true fix for all of these DNS spoofing attacks (Kaminsky, Zodiac, etc.) is to implement DNSSEC, which creates a "chain of trust" throughout the DNS hierarchy and uses digital signatures to validate DNS responses. More on this later.

Split-Horizon DNS

Split-Horizon DNS discusses the theory behind presenting one version of your DNS information to the outside world and a completely different view internally—why and when this is useful and some architectural issues with such a configuration.

What Is Split-Horizon?

- One version of DNS for outside, keep different DNS inside
  - Outside gets “bare-minimum” information
  - Inside sees complete set of DNS records
- Implies running two different sets of name servers
- Can be combined with DNS proxying

As discussed earlier, giving away too much DNS information to the outside world can help attackers map your networks or choose vulnerable or otherwise "interesting" machines to target for initial attacks. *Split-horizon* (sometimes called *split-brain*) *DNS* is a DNS configuration where an organization presents one set of DNS information to external organizations and reserves a second, separate set of DNS information for internal use. This is generally done by maintaining two different collections of name servers: An "external" set that publishes the limited amount of DNS information external organizations need to interact with your company, and an "internal" set that holds your complete, rich set of DNS information. Note that while the separate DNS zone databases are generally maintained on two different sets of physical machines (and this is the configuration we shall describe in this course), it is possible under BIND v8 to run to different name servers with different zone databases on the same machine.

When your internal name servers wish to resolve external host names, they must contact root name servers and name servers at other internet-connected sites. This can open up your internal name servers to attack from the outside. For this reason, many organizations that run split-horizon DNS also employ a sort of DNS proxying (*slave forwarding name servers*) to "hide" their internal name servers completely from the outside world.

Digression: IP-Based Auth …

- Remote server receives IP source address
- Server does a DNS lookup to map IP address to hostname
- Server then looks up hostname to get IP address
- If addresses don't match, access is denied

One common simple form of authentication used by some internet servers (often FTP daemons, and services protected with TCP Wrappers) uses a combination of reverse (IP to hostname) and forward (hostname to IP) DNS lookups. A remote server will take the IP address it receives as the source of a connection and reverse that address to a hostname. Looking up this hostname yields an IP address. If the original address received as the source of the connection doesn't match the IP address retrieved from DNS, then the remote server assumes the connection comes from an attacker trying to spoof IP addresses and/or DNS information and will not allow the remote machine to connect.

… Split-Horizon Implications

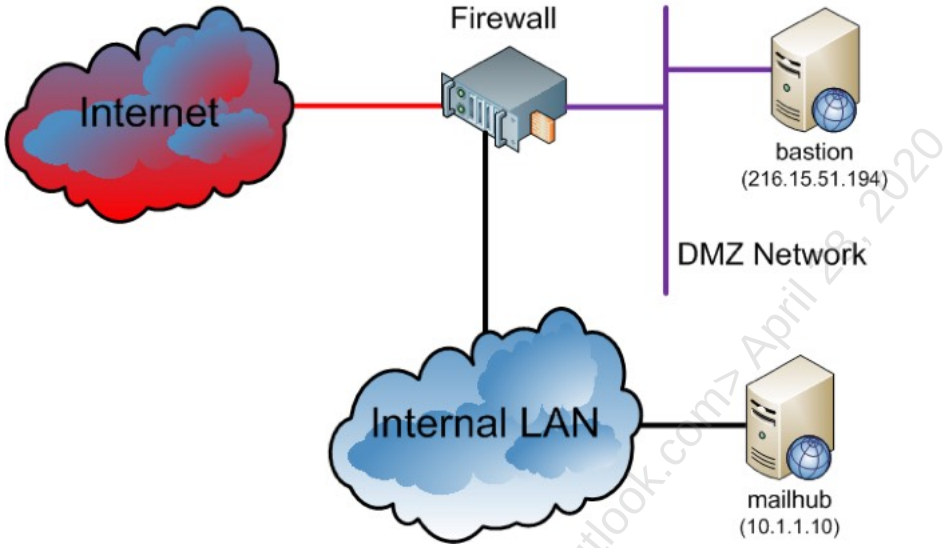
- Must advertise all hosts with internet-routable IPs
- Thus, split-horizon useful only with NAT or proxy servers
- Could use dummy records if all hosts must be advertised

The implication of this form of authentication is that sites should maintain both forward and reverse DNS information for all hosts that are capable of connecting directly to the internet. This information must be made available in your "external" DNS database in a split-horizon configuration.

Unfortunately, if all of your hosts can connect directly to the internet, that means you have to advertise information about all of your machines in your external split-horizon database—largely nullifying the usefulness of a split-horizon configuration. On the other hand, if your organization uses Network Address Translation (NAT) or proxy servers, you generally only have to advertise information about your NAT pool or proxy servers and split-horizon is useful.

Note that even if you have to advertise information about all of you internal hosts, you don't need to tell the outside world your HINFO or TXT records, so split-horizon can be somewhat useful. It is also sometimes useful to present one set of mail routing information (MX records) to the outside world and have a different configuration internally.

Typical Firewall Architecture



Here is a diagram of a simple, but fairly typical firewall configuration in use at many organizations today. The organization has a multi-legged firewall which connected the external internet to both a semi-private de-militarized zone (DMZ) network and a private internal corporate network. The DMZ network is where an organization would put its web and FTP servers and any other machines that the outside world needed to reach—and for purposes of this example, a machine called `bastion` that will be the "external" DNS server for our split-horizon example (and also the external email relay for the Appendix on *Sendmail* configuration). On the internal network, there is a machine `mailhub` that acts as the primary server for the "internal" DNS (and will be the internal mail server in the Appendix).

Note that combining DNS and email services on a single server is *not* recommended practice since a security vulnerability in one service will quickly become an issue for the entire system. I'm combining services in this example simply to reduce the number of machines in play and make things less confusing. But in the real world, you should run your DNS servers and email servers on separate machines (or at least separate VM instances).

Most Restrictive Assumptions

All inbound/outbound traffic must stop on DMZ

- Internet hosts can't talk to internal machines
- Internal hosts can't reach internet hosts
- `mailhub` can reach `bastion`
- `bastion` has a hardened OS
- `mailhub` may be hardened

For purposes of example, we shall assume the most restrictive possible firewall configuration—assuming you have a working configuration for this environment, you can easily adapt the configuration for your own (less restrictive) firewall setup. In particular, we assume that hosts on the internet can only route packets to hosts on the DMZ network and that the DMZ is the only network that is allowed to communicate with the internal corporate network. Internet machines are not allowed to communicate with "internal" machines and vice versa.

It's a good idea to spend effort removing dangerous services (read: Everything not absolutely needed for mail and DNS transport) from the `bastion` host. Everybody in the world will be trying to break into this system. Given the chance for email address (stack smashing) attacks that can work inside machines even through the bastion host, tightening down your internal mail servers is also probably a good idea.

DNS: Goals

`bastion` is DNS for external hosts:

- Contains limited zone information
- MX records to force mail to *bastion*

`mailhub` is internal name server:

- Contains richer set of information
- Internal-only subdomains may exist
- Must proxy outgoing DNS requests

Again, the idea is that the external DNS for your organization will contain the bare minimum necessary for your organization to successfully conduct business on the internet. It should not advertise internal hostnames, hardware or operating system information (HINFO and TXT records), and the like since this information could be useful to crackers trying to penetrate your network.

The internal DNS information at your site will contain complete information for your domain including useful host aliases, subdomain information, and system information. This DNS information will only be visible to hosts owned by your organization.

However, since the internal DNS servers are unable to talk to the internet, we must arrange some sort of DNS proxy solution to resolve external names like *www.sans.org*.

Configuring BIND

Configuring BIND presents specific configuration examples of the DNS architecture introduced in the *Split-Horizon DNS* section, and introduces many of the new security features in recent releases of BIND.

bastion-named.conf

```
options {
    directory "/etc/namedb";
    version "like nothing you have ever seen";
    allow-transfer { 207.90.181.1; 207.90.181.2; };
    allow-recursion { 10.1/16; 216.15.51/24; };
    allow-query-cache { 10.1/16; 216.15.51/24; };
    rate-limit { responses-per-second 10; };
};

zone "." {
    type hint;
    file "named.ca";
};
```


This is the new BIND configuration file syntax that was actually introduced in BIND v8 (and is still used in BIND v9). The Syslog-NG developers essentially copied the BIND `named.conf` syntax for the `syslog-ng.conf` file, so you'll see a lot of similarities in the configuration file syntax between the two programs.

The `options` block applies globally to the rest of the configuration. The `directory` statement means that all other filenames are relative to `/etc/namedb` (e.g. `/etc/namedb/named.ca`). In fact, `directory` actually changes the value of `DESTRUN`, the default working directory for BIND.

As demonstrated earlier, it is possible for outsiders to query your running name server and find out what version of BIND you are running. Since certain versions of BIND have known vulnerabilities, you want to hide what version your name servers are running. The `version` option allows the administrator to specify an arbitrary string instead of the actual BIND version number. In this example, the attacker will be able to guess you're running at least BIND v8 (the first release where the `version` option appeared), but nothing more specific than that. Theoretically, you could change the version string to be a valid version string for an earlier version (e.g. `"4.9.7"`) to confuse attackers.

On your master DNS server, you should use the `allow-transfer` option to restrict zone transfers to only those machines that are legitimate secondary servers for your domains. If you do not specify an `allow-transfer` list, then any server on the internet may do zone transfers from your server.

But as we discussed earlier, slave DNS servers are also capable of servicing zone transfer requests, so we need to specify an `allow-transfer` list on these servers as well. What list should you use? Some sites simply configure the same canonical list of servers on all machines, so any machine will be willing to provide zone transfers to any of the others. You also have the option of specifying "`allow-transfer { none; }`" on your slaves to completely prevent zone transfers from these machines. Personally, I like to put just the IP address of the master server in the `allow-transfer` block on my slaves—that way, as the administrator of the master server, I can request zone transfers from my slaves when I need to confirm that they've got the latest version of the zone files, but I still prevent unauthorized machines from doing zone transfers.

Earlier, we discussed the problem of allowing anybody to make recursive queries and/or cache queries at your DNS server—you'll most likely end up being used as an amplifier in a DNS DoS attack and the performance of your own DNS servers may be significantly affected. The `allow-recursion` and `allow-query-cache` settings allow you to specify a list of netblocks that should be allowed to perform these functions against your name server. In general, you should only list the netblocks of networks that are under your control and never allow external networks to make these sorts of queries.

In fact, if you have a name server that *only* exists to respond to queries from the outside world—as is often the case if you're doing split-horizon DNS—you may not need to allow either recursive queries or cache queries to anybody. In this case, you can use a configuration like:

```
allow-recursion { none; };
allow-query-cache { none; };
```

In our case, however, we are expecting to use the bastion server as a proxy for DNS requests from the internal network, so we need to allow recursion and cache queries from these networks.

BIND 9.9 introduced the `rate-limit` option to further reduce the opportunities to use BIND for DDoS attacks. For more details on tuning and additional options, see *https://kb.isc.org/article/AA-00994/0*

Note that `version`, `allow-transfer`, `allow-recursion`, `allow-query-cache`, and `rate-limit` options are most appropriate for your external name servers. You can apply them to your internal name servers as well, but you may find this more of an administrative hassle than anything. Aside from having to maintain the lists of IP addresses in the `allow-transfer` and `allow-recursion` options, it is occasionally useful to be able to query your own internal name servers and find out what version of BIND they're running (so you know which ones to upgrade).

The latest version of the `named.ca` file is available from the InterNIC as

`https://www.internic.net/domain/named.root`

Alternatively, you can actually just run "`dig @a.root-servers.net . NS >named.ca`" to produce the necessary file.

More of `named.conf` on `bastion`

```
zone "sysiphus.com" {
    type master;
    file "sysiphus.hosts";
};

zone "51.15.216.in-addr.arpa" {
    type master;
    file "sysiphus.rev";
};
```


Now, for each zone, we specify whether this name server is the `master` server for the domain (sometimes referred to as a *primary* server by DNS administrators) or a `slave` (*secondary* server). Then we give a filename (again, relative to `/etc/namedb`) where the zone data can be found.

The `in-addr.arpa` domain is where you maintain the mappings between IP addresses you own and hostnames. Read the domain from right to left—we are saying that bastion is the master server for all addresses in the `216.15.51.0` network.

Note that `sysiphus.com` is an example domain used in this course. Be sure to replace it with your local domain name when you get back to the office!

bastion- sysiphus.hosts

```
; Local domain configuration for external DNS
$TTL 1d
@     IN SOA bastion.sysiphus.com. hostmaster.sysiphus.com. (
       2019050100   ; Serial - year/month/date/revision
       1d           ; Refresh from server
       1h           ; Retry after failure
       7d           ; Expire data
       2h )         ; Negative cache TTL

@                IN      NS      bastion.sysiphus.com.
                 IN      NS      ns6.gandi.net.
                 IN      MX 10   bastion.sysiphus.com.
*                IN      MX 10   bastion.sysiphus.com.
```


The SOA ("Start of Authority") record defines global properties for your domain and lists contact information for the domain (hostmaster.sysiphus.com instead of hostmaster@sysiphus.com since @ is a restricted character).

Generally speaking, the values in the SOA record above are appropriate for sites that don't make a great deal of DNS updates—your external DNS should be relatively static. Don't forget to increment the serial number value every time you make changes to your DNS information (here, we're using a date-based serial number: The last two digits are provided in case you make more than one update on a given day).

You should always have at least two advertised name servers for your domain, just in case the network partitions. If these hosts can be situated on two different major service provider networks, so much the better. Note that the name server listed above belongs to the author's registrar—please do not advertise this name server as slave server for your domain!

The MX ("Mail Exchanger") records are sending all mail for sysiphus.com and *.sysiphus.com to bastion. Wildcard MX records are dangerous if you're not using split-horizon DNS, but administratively convenient unless you have some automated mechanism for maintaining your DNS tables.

sysiphus.hosts

```
; host info below

bastion     IN    A          216.15.51.194
ns          IN    CNAME      bastion
mail        IN    CNAME      bastion

server      IN    A          216.15.51.195
www         IN    CNAME      server
ftp         IN    CNAME      server
```


Here are some standard forward address records that might appear in your external DNS information. A `CNAME` (*canonical name*) record is a mechanism for setting up an alias for a given machine. Note that we use `CNAME`s to associate functional names with specific machines—we could just as easily have configured multiple `A` records all pointing at the same IP address. While religious debate runs hot and heavy about whether `CNAME`s or `A` records should be used in these situations, the reality is that *it fundamentally doesn't matter*—there is no substantial technical reason to prefer one over the other.

bastion-sysiphus.rev

```
; Reverse address lookups for external DNS
$TTL 1d
@     IN SOA bastion.sysiphus.com. hostmaster.sysiphus.com. (
      2019050100   ; Serial - year/month/date/revision
      1d           ; Refresh from server
      1h           ; Retry after failure
      7d           ; Expire data
      2h )         ; Negative cache TTL

@               IN      NS      bastion.sysiphus.com.
                IN      NS      ns6.gandi.net.
195             IN      PTR     server.sysiphus.com.
194             IN      PTR     bastion.sysiphus.com.
```


This file is very similar to the `sysiphus.hosts` file, except that it contains `PTR` records instead of `A` and `CNAME` records. Note that the leftmost column of the `PTR` record entries reverses the order of the octets of the hosts' addresses. The trailing dot at the end of the `PTR` record is vitally important.

mailhub– named.conf (I)

```
options {
    directory "/etc/namedb";
    forwarders { 216.15.51.194; 216.15.51.194; };
    forward only;
};

zone "." {
    type hint;
    file "named.ca";
};
```


Remember that we are assuming our internal name server is unable to reach internet-connected hosts for DNS information. The `forwarders` lines cause the internal name server to send all external DNS queries to `bastion` to be resolved. In the event the query fails, the local server would normally attempt to contact a remote name server, but the "`forward only`" directive prevents this behavior.

Note that we are listing the IP address of `bastion` twice in the `forwarders` directive. The timeout on `forwarders` queries is actually shorter than the timeout on the normal DNS queries made by the external machine `bastion`. Hence, we list the IP address of `bastion` twice so that the local server will retry in the event it fails to get a response to its first query.

In the "real world" of course, you would probably have multiple external name servers to forward queries to in order to provide some redundancy. If you had name servers A, B, and C to list in your `forwarders` directive, you are probably better off listing their IP addresses `{ A; B; C; A; B; C; }` rather than `{ A; A; B; B; C; C; }`. After all, the most likely reason to not get a response to your first query is that machine A is down, so you want to try another name server first before retrying server A.

All other configuration statements in this part of the file are the same as they were on the `bastion` host.

mailhub– named.conf (2)

```
zone "sysiphus.com" {
    type master;
    file "sysiphus.hosts";
};

zone "1.10.in-addr.arpa" {
    type master;
    file "sysiphus.rev";
};
```


This section of the file defines the zone files for the internal copy of the `sysiphus.com` zone and the `in-addr.arpa` domain for the internal network (remember the zone file names are relative to `/etc/namedb`). The main drawback to split-horizon DNS is that some information is necessarily duplicated between the internal and external versions of the `sysiphus.com` zone—primarily address information for the external "public" servers. Some sites choose to develop scripts that automatically dump out the "external" zone information from the internal zone files in order to avoid mistakes where one copy of the information is updated and not the other.

mailhub– named.conf (3)

```
zone "eng.sysiphus.com" {
    type slave;
    file "eng.zone";
    masters { 10.1.64.254; };
};

zone "corp.sysiphus.com" {
    type slave;
    file "corp.zone";
    masters { 10.1.128.2; };
};
```


In this section of the file, we see some internal domains that are not known to the outside world. In this case, `mailhub` is a `slave` server of these domains and gets its zone information from some other internal `master` name server.

Remember, the zone file names are relative to `/etc/namedb`.

Lab Exercise

Running BIND `chroot()`ed

You'll learn this better if you do it by hand

Make sure:

- SELinux is in Permissive mode
- You ignore pre-configured `chroot()` area

Exercises are in HTML files in two places: On the course USB media, and in the home directory of the SANS user in the lab virtual machine. These two locations contain identical copies of the exercise files—it just depends on whether you prefer to look at them from a web browser inside the lab VM (because you're working in full-screen mode) or from your host operating system.

1. Open your web browser
2. Select "File… Open…" from the browser menu (Ctrl-O is the usual keyboard shortcut)
3. This is Day 5, Exercise 1, so navigate to *.../Exercises/Day_5/index.html* and open this HTML file
4. Click on the hyperlink, which is the title of Exercise 1
5. Follow the instructions in the exercise

DNSSEC

Earlier, when we were talking about different kinds of DNS spoofing attacks, I mentioned that the "best" fix for these problems was to implement DNSSEC. But I also said that DNSSEC is "hard" on many levels. So, I'm going to show you how DNSSEC works and give you some tools for implementing it in your organization.

Before we begin, let me say that the information in this section is based on the DNSSEC implementation BIND 9.4.x and later, and discusses functionality introduced in v9.7.x and v9.9.x. If your OS includes an older version, you'll need to upgrade in some fashion—possibly by building BIND from the source distribution.

Good Idea In Theory

All DNS responses digitally signed

"Trust" comes through DNS hierarchy:
- Parent domains sign keys for children
- Keys for root zone distributed out-of-band

Should defeat all current spoofing attacks

The basic idea behind DNSSEC is reasonably straightforward:

- You generate a key pair and use it to sign your zones.
- You pass your public key back up the DNS chain to the owner of your TLD—for example, if we're `sysiphus.com`, then we give a copy of our key to the registry for `.com`.
- The TLD uses DS (Delegation Signer) records to publish your public key (actually the key fingerprint) and then signs those records with its key.
- Similarly, the root zone maintainers use DS records for all of the TLDs and sign those keys with the key for the root zone.
- Everybody manually downloads the key for the root zone from central authority and verifies the key using something like a PGP signature. Think of this as being similar to the current process of locally caching the names and IP addresses of the root name servers.

Once this infrastructure is in place, then you should be able to verify DNS responses because there will be a "chain of trust" from the root of the DNS hierarchy. Attackers will be unable to spoof DNS responses unless they manage to steal the private key for some zone (so we'll need to rigorously protect those keys).

Current Implementation? Well …

Hassle for local zone admins:

- Zone signing: Security vs. convenience
- By default, keys recycle every 30 days

Political/technical issues at the root:

- TLD issues already contentious, now you're adding PKI…
- Need process for registrars to manage keys

The original DNSSEC implementation required administrators to manually sign the entire zone file for every update, even if that update was only for a single record. This architecture allowed you to keep your signing keys off-line for better security but was a burden on administrators. BIND 9.7.x introduced the ability to keep the signing keys on your name server and sign zones on-the-fly. The downside is that an attacker who compromises your name server can now steal your keys.

By default, in the current implementation, the signatures you generate are only valid for 30 days. Furthermore, the recommendation is that you generate a new key to sign your zones at each 30-day interval (this is referred to as a *rollover* in most DNSSEC documentation). DNS caching comes into play and you typically end up having multiple keys in play per zone to make everything work out. More on this later.

Beyond the problems for individual domains, there are all kinds of political and structural issues when it comes to implementing DNSSEC at the root and in the TLDs. Consider that all of the TLD maintainers and registrars are going to have to work out a scheme to get keys propagated up from the people who are registering zones and shoved into the appropriate zone databases, and then get those TLD zone databases signed. That's a big infrastructure issue for these companies.

Also, *anything* you do at the top of the DNS hierarchy is going to offend somebody, it seems. Other countries already feel that the US has too much control over the DNS infrastructure, and there is an ongoing political battle about managing keys in the top-level domains.

Understanding Key Types

Zone Signing Key (ZSK):

- Ephemeral key used to sign zone records
- This is the one that cycles every 30 days

Key Signing Key (KSK):

- Long-lived "trust" key used to sign ZSKs
- This is the key your parent domain signs

There ends up being two different types of keys running around when you start using DNSSEC. First, there are the *Zone Signing Keys (ZSK)*. These are the keys that are actually used to sign your zone files. But these are also the keys that you should recycle every 30 days. It would be awful if you constantly had to be rolling these keys over and feeding the new keys back up to your TLD registrar. In fact, it would be totally unworkable in practice.

So, the recommended strategy is to also create a *Key Signing Key (KSK)*. As the name implies, you use the KSK to sign your ZSKs. You pass the public portion of the KSK up to your TLD registrar and that's what gets signed and put into DS records in the TLD. Basically, you're setting up a "chain of trust again"—the world trusts your KSK because it's signed by the TLD, therefore they can trust your ZSKs that are signed by your KSK, and therefore they can trust the individual zone records signed by your ZSK. The KSK is supposed to last for a long time—more like the multi-year lifetimes of SSL certificates.

So, why bother with all of this? Why can't we just keep the ZSKs around for longer? I assume that the thought process is that with sufficient access to a variety of DNS queries and responses, it might be possible to eventually mount an attack on the private portion of the ZSK. So, the ZSK is cycled before such an attack would be computationally feasible. Since the KSK is only used to sign a limited number of ZSK records that change relatively slowly, there's less material an attacker could use to break this key, prolonging its useful lifespan.

Signing Your Zones

1. Generate KSK and ZSK(s)
2. Include new keys in zone file
3. Generate signed zone files
4. Update `named.conf` file

First, we're going to cover the basic process of manually creating signed zone files. This includes generating keys and incorporating them into your zone files and then actually signing all the records in the zone file. Then there will be some tweaks you have to make in your `named.conf` file to turn on DNSSEC and use the new signed zone file.

Once we have the signed zone files all squared away, we'll look at other issues like joining into a "chain of trust" with the rest of the DNS hierarchy and also issues that come up when you have to "rollover" your ZSK. More on this in just a moment.

Step 1: Generating Keys

```
# dnssec-keygen -r /dev/urandom -f KSK \
                          -a RSASHA256 -b 2048 sysiphus.com
Ksysiphus.com.+005+21946
# mv Ksysiphus.com.+005+21946.key Ksysiphus.com.ksk.key
# mv Ksysiphus.com.+005+21946.private Ksysiphus.com.ksk.private
```

- Use `/dev/urandom` unless you feel like waiting...
- Generate ZSK by dropping "`-f KSK`"
- Make two ZSKs: "`curr`" and "`next`"

The `dnssec-keygen` command (comes with BIND) is used to generate both ZSKs and KSKs. The only difference is whether or not you include the "`-f KSK`" option on the command line. I'll also note that by default, `dnssec-keygen` uses `/dev/random` to generate entropy, which can take a long time for large keys. I generally use `/dev/urandom`, which only generates pseudo-random numbers, but which is "good enough" in practice.

`dnssec-keygen` creates filenames like `K`*<domain>*`.`*<gibberish>*`.*` (the `.key` file is your public key, the `.private` is the secret half—protect this file!). I personally prefer to rename my files, replacing *<gibberish>* with a useful name that indicates which key is which. Notice that `dnssec-keygen` outputs "basename" of the file it's creating, without the `.key` or `.private` extensions.

For reasons that will become clear when we start talking about key rollover, you should actually create two ZSKs at this point. One will be our "current" ZSK, which we use to sign our zone files. The other will be the "next" ZSK we use when it's time for the rollover. So, your commands for creating and renaming these keys might look like:

```
# dnssec-keygen -r/dev/urandom -a RSASHA256 -b 2048 sysiphus.com
Ksysiphus.com.+005+17565
# mv Ksysiphus.com.+005+17565.key Ksysiphus.com.curr.key
# mv Ksysiphus.com.+005+17565.private Ksysiphus.com.curr.private
#
# dnssec-keygen -r/dev/urandom -a RSASHA256 -b 2048 sysiphus.com
Ksysiphus.com.+005+11811
# mv Ksysiphus.com.+005+11811.key Ksysiphus.com.next.key
# mv Ksysiphus.com.+005+11811.private Ksysiphus.com.next.private
```

Step 2: Update Zone Files

```
# cat >>sysiphus.hosts
$include Ksysiphus.com.ksk.key
$include Ksysiphus.com.curr.key
$include Ksysiphus.com.next.key
^D
```

- Make sure you use the `*.key` files
- If keys in separate directory, use correct path names

If you look at the `*.key` file that's generated by `dnssec-keygen`, you'll see that it's actually formatted as a `DNSKEY` record:

```
# cat Ksysiphus.com.ksk.key
sysiphus.com. IN DNSKEY 257 3 8
AwEAAQFyb9DzHm2siAd62P+6jtaqdRp26ZPn3cYHrVEL5f5noqBLoVHh
5EuCazU1OOccErdLZu/ZPEyOl2fL3Qv+QqeUpuG1CJpaZhfZkYHuM98Y
ZxmoV3SGOc4YGJci7knEddnzpfz6MgUSMehRfKCV9csr8GQ7QwmTF5KK
JVcZNdu+6lR6z1Ku8Aym9uywD4j9s88Xy0ym/ZFttDDPq6HQsnPWZabH
JYwndasEawKIn3tQhS9oGHDqsHOODk373ETDBwhvMYtrXDxYfmLTikI8
ptFCqGZVW1GdayHpqklTNe3EI4naGzDUgwG0pBmhEiDJk0fzcRkVwDl…
```

You need to include these records in your zone file so that they can be signed with the rest of your zone. You could cut 'n' paste them directly into your zone files, but since the ZSKs will be changing regularly, keeping them in separate files is actually a lot easier—especially once you start automating the rollover process. The zone-signing process automatically sucks the contents of the included files into the final, signed zone file so you end up with only one file you need to copy into the directory used by your named process.

If you have a lot of signed zones, you'll find that keeping the keys and the zone files in the same directory becomes too cluttered. Personally, I keep all my key files in a separate directory, so my directives look more like "`$include ../keys/Ksysiphus.com.ksk.key`", etc.

Step 3: Signing Zone Files

```
# dnssec-signzone -o sysiphus.com -k Ksysiphus.com.ksk \
      sysiphus.hosts Ksysiphus.com.curr
sysiphus.hosts.signed
```

- Yes, you have to do this every time you make a zone update
- Warning: Signed zone is MUCH larger than unsigned zone
- Also creates files for upstream registrars

Once you have your keys and you've updated your zone file with the "`$include`" statements, you're ready to use `dnssec-signzone` to produce a signed zone file. The "`-o`" option specifies the "origin", i.e. the name of the domain you're signing. Use "`-k`" to specify the KSK—notice that we're using the "basename" of the file without the `.private` extension. Then you specify the filename to work on and the ZSK that should be used to sign the file (again, use the "basename" without the extensions). `dnssec-signzone` outputs the name of the signed file—by default, this is <*filename*>`.signed`, where <*filename*> is the original filename specified on the command line.

Remember that you need to *regenerate* the signed zone file every time you make updates to the `sysiphus.hosts` file. Yes, this sucks. We'll look at some ways to automate this in just a moment.

The other bit of bad news is that your signed zone file can be more than 10x larger than your unsigned file. This translates to much higher memory usage in your name servers. Maybe this isn't a big deal for a small company with just a few zones. But for an ISP, hosting provider, or a TLD, this can be a huge issue.

Aside from the signed zone file, `dnssec-signzone` also generates a couple of other files. The `keyset-`<*domain*>`.` file contains a `DNSKEY` record for the public half of the KSK for the zone. The `dsset-`<*domain*>`.` file contains `DS` records. Normally, you would pass this information upstream to your TLD registrar for them to include in their zone files and sign with their key.

Step 4: named.conf Updates

```
options {
        …
        dnssec-enable yes;
        dnssec-validation yes;
};

managed-keys {
        . initial-key 257 3 8 "AwEAAagAIKlV…
};

zone "sysiphus.com" {
        type master;
        file "/master/sysiphus.hosts.signed";
};
```


OK, you've got your signed zone file. Now what?

All of the name servers (primary *and* secondary) will need to add the `dnssec-enable` and `dnssec-validation` options in their `named.conf` file. On the primary server, you also will typically have to change your `zone` declaration to use the signed file instead of the old, unsigned version.

At this point, you've established what the docs generally call an "island of trust." You've got DNSSEC enabled in your name servers and you're publishing a signed zone file. But nobody else trusts your responses until you push your key info up through your registrar and get it signed. You also need to add the key for the root zone via the `managed-keys` block. You can obtain the necessary key from *https://www.iana.org/dnssec/files*

But how can you validate that DNSSEC is actually working? There are two mechanisms you can try:

1. "`dig +multiline +cd +dnssec +trace sans.org`" and you'll see all the DNSSEC validation records in the output. Try other domains to see what things look like when there is no DNSSEC enabled.
2. Alternatively, you could add this to your `named.conf`:

```
logging {
  channel dnssec_logs {
            syslog local4;
            severity debug 3;
  };
  category dnssec{ dnssec_logs; };
};
```

Configure Syslog to put `local4` logs into a separate log file, `HUP` your name server, and try those `dig` queries again. You'll see all the gory detail of the DNSSEC traffic.

Internal Testing Only

```
options {
        …
        dnssec-enable yes;
        dnssec-validation yes;
};

trusted-keys {
        sysiphus.com. 257 3 8 "AwEAAQFy…RI24IcH";
};
```


You can also set up hard-coded trust within your environment for testing purposes. Since you have control over these machines, you can manually copy the public half of your KSK to each system and directly load it into the `named.conf` file on these systems.

As you can see on the slide, the `named.conf` configuration directive is `trusted-keys`. This is where you list the public info for the KSK of any zone(s) you want to be able to validate. You can pull this information out of the `K`*<domain>*`.ksk.key` file we created with `dnssec-keygen` earlier. These entries are long and cumbersome, so if you end up with a lot of them, you'll probably put them in another file and just "`include`" them into `named.conf`.

Once you've got `trusted-keys` configured and turned on `dnssec-enable` and `dnssec-validation`, your resolver will start validating responses from the zones you configured in `trusted-keys`. It will also *stop* accepting unsigned responses from these zones (and will obviously drop responses with incorrect signatures).

Key "Rollover"

Signatures only good for 30 days

Should use new key each interval

DNS caching issues:

- Can't just remove old ZSK
- Publish new ZSK *before* signing can start
- Need to do this well before 30-day limit!

This is why we made two ZSKs

At this point, we've seen how to sign a zone file and join in the DLV experiment. However, the real fun starts after the first 30 days and we need to do a key rollover in our zone(s). Actually, you'd better start this process well in advance of the 30-day limit.

DNS caching combined with DNSSEC forces us to be a lot more deliberate in our rollover actions. You can't just drop your old ZSK and replace it with a new one. If we did this, we'd run into problems in two ways:

1. It's possible that another site could have cached one of your DNS records with the signature from the old ZSK, but have timed out the DNSKEY record that corresponds to the old ZSK. This can force a situation where the remote side queries your name server for the DNSKEY record for the new ZSK. Since the cached signature doesn't match the new ZSK, the cached record is marked invalid on the remote side and SERVFAIL is returned. The remote name server will *not* try and look up the invalid record again, so this situation results in denial-of-service.
2. The opposite case is also possible—namely the remote site is caching the DNSKEY record for your *old* ZSK and then tries to look up a record in the current zone that's signed by the new ZSK. Again, failure and denial-of-service is the result.

The fix for problem #1 is to keep the DNSKEY record for the old ZSK around for some period of time after you've switched over to use the new ZSK to sign the zone. At least remote sites will still be able to find the old ZSK to validate cached signatures. The fix for problem #2 is to publish the new ZSK well before you start using it to sign zones. That way, it will be cached in remote name servers before the rollover.

So now you understand why I had you create two ZSKs earlier. When it's rollover time, the "`curr`" key becomes the "`old`" key, and the "`next`" key becomes "`curr`". When I do this swap, I also create a new key to be the "`next`" key. So, I normally actually have three ZSKs running around for each zone. Just to be extra safe, I also generally cycle keys every 14 days. Really, your timeout window is something more like "(30 days) —(maximum time to live)", but cycling the keys faster than that gives you some leeway with broken resolvers that cache information longer than they should.

Let's See That In Code …

```
# rename curr old Ksysiphus.com.curr.*
# rename next curr Ksysiphus.com.next.*
# dnssec-keygen -r/dev/urandom \
    -a RSASHA256 -b 2048 sysiphus.com
Ksysiphus.com.+005+22785
# rename +005+22785 next Ksysiphus.com.+005+22785.*
# echo \$include Ksysiphus.com.old.key >>sysiphus.hosts
# dnssec-signzone -o sysiphus.com -k Ksysiphus.com.ksk \
    sysiphus.hosts Ksysiphus.com.curr
sysiphus.hosts.signed
```

"Hmmm, we should automate this ..."
"Oh really? You think?"

Here are the commands you would use to implement the rollover strategy I described on the last slide. First, the "`curr`" files get renamed to "`old`" and the "`next`" files are renamed to "`curr`". Then I generate a new ZSK and call it the "`next`" key. We actually don't yet have a `$include` directive in our zone file for the "`old`" key, so I add that.

Finally, I sign the zone file with the "`curr`" key, just as we did before. However, this time the "`curr`" key was what was formerly the "`next`" key.

Even if you do this once in a blue moon, it's likely that you'll make a typo and mess something up. And there's no way you're going to do this every two weeks by hand. So, you automate the process, but really this doesn't have to be that difficult…

Simple Automation Strategy

Main automation points:

1. Generating initial KSK and ZSKs
2. Handling regular rollover cycle
3. Signing zones during regular updates

Wrote `genkeys` script to handle #1-2

`Makefile` handles #3 (automatically calls `genkeys` to do #1)

Really, there are three things we need to automate here: Generating the initial keys for a zone, rolling the ZSKs on a regular basis, and automating the zone file signing process for both rollovers and for normal zone updates. I've included a couple of tools on your course DVD to make this easy.

First, there's the `genkeys` script. If you call it with a domain name as a command-line argument, then it will generate new keys for that zone—creating a KSK if one doesn't exist as well as cycling the ZSKs or creating new ones if they don't exist. If called with no arguments, it will look in its working directory (defined on the command line or in the script itself) for any keys that are 14 days old or older and cycle the keys for the associated domain. This latter mode means that `genkeys` suitable for calling from `cron` on a daily basis to automatically cycle your keys.

I've also included a `Makefile` for the directory where you keep your zones. It looks for zone files named <*domain*>`.db` and creates `../signed/`<*domain*>`.signed` files. It will even call `genkeys` automatically if there are not currently any keys for the domain.

Similarly, `genkeys` will automatically run a "`make install`" in your zone files directory every time it ends up rolling keys for any of your zones. "`make install`" copies the newly signed zone files into place and `HUP`s your `named` process.

When the `Makefile` calls `dnssec-signzone`, it uses the "`-N unixtime`" option to update your SOA serial number automatically, so you no longer need to worry about that. It needs to be able to do this automatically so that `genkeys` can handle ZSK rollovers without human intervention. The only problem is that the "`unixtime`" (number of seconds since Jan 1, 1970) may be smaller than your current serial number. See this URL for more information on how to roll your SOA serial number over to a smaller value:

https://fatmin.com/2011/01/21/bind-zone-file-serial-number-reset/

Summary

- Important to be able to trust DNS
- Admin interface is improving
- Coming soon?

We really need to get DNSSEC implemented for security reasons. Unfortunately, there are some technical, administrative, and political hurdles that need to be overcome before this happens. But if nobody wants to use DNSSEC because it's too hard for sites to implement, then we're never going to be able to tackle the bigger issues.

Lab Exercise

- DNSSEC and you
- Experiment with keys and zone signing

Exercises are in HTML files in two places: On the course USB media, and in the home directory of the SANS user in the lab virtual machine. These two locations contain identical copies of the exercise files—it just depends on whether you prefer to look at them from a web browser inside the lab VM (because you're working in full-screen mode) or from your host operating system.

1. Open your web browser
2. Select "File… Open…" from the browser menu (Ctrl-O is the usual keyboard shortcut)
3. This is Day 5, Exercise 2, so navigate to *.../Exercises/Day_5/index.html* and open this HTML file
4. Click on the hyperlink, which is the title of Exercise 2
5. Follow the instructions in the exercise

Apache Web Server

This section covers basic security configuration information for the Apache web server (by far the most popular web server on the Unix platform).

This section does not attempt to address security issues with server-side application environments like Tomcat, WebSphere, etc. Consult appropriate vendor documentation in these areas. You may also find the following "Tech Tips" from the CERT/CC useful when doing server-side programming:

http://www.cert.org/tech_tips/malicious_code_mitigation.html

Biggest Security Problems

Poorly written CGI/PHP scripts and other "server-side" code

Buffer overflows

Poor configuration practices:

- Running servers with privilege
- Running "helper apps" with privilege
- Overlapping web root with other data/apps
- Programs, libraries, config files in doc root

Before we start talking about how to secure Apache web servers, it's useful to look at some of the most common ways web servers are compromised.

One common issue is not a web server security issue per se, but rather the fault of badly written (or in some cases maliciously written) code that is run by the web server. Complete coverage of secure server-side programming could take an entire week-long course (or more), but the short answer is disable server-side execution if you're not using it and make sure to carefully review any server-side code you are running. Also, take steps to protect the software that's being run by the server and keep it out of the normal document trees (more on this later).

The Apache web server has had its share of buffer overflow problems. Keeping up-to-date on software releases is the primary defense here. Of course, you may be vulnerable to a previously undiscovered exploit in the current code, so some sites will run their web servers in a `chroot()` environment (the Apache Benchmark document at *CISecurity.org* provides an example). However, `chroot()`ing a complex Apache install with lots of CGI scripts and other server-side executables can be difficult.

Most of the other common Apache exploits come down to poor administration and configuration. There should never be a reason to run the web server or any other apps related to the server (like CGIs, MySQL servers, etc.) with root privileges—all of these processes should be running as unique unprivileged UIDs. Another common mistake is to overlap the web document root with the data area of another server—for example, sites that mix their web docs with an anonymous FTP upload/download area. This opens a window for security problems in one service to impact the other, or possibly sets up a way to turn small vulnerabilities in each service into a massive problem for the machine as a whole. Another common mistake is to put executable code and/or server configuration files into the document root (some sites have accidentally put server password files and site certificates in vulnerable locations!). This may allow external attackers to view the source code for your apps (making it easier to spot vulnerabilities in the software).

apache.org Compromise

Jan. 2001 *apache.org* compromise is a perfect example:

- Web root and anon FTP area overlapped
- Uploads to FTP area handled insecurely
- Web server allowed server-side execution
- MySQL server running as root

Result was cute defacement; could have been much worse ...

The January 2001 break-in at `apache.org` is a good example of the kinds of configuration mistakes we were discussing on the previous slide:

- At the time of the incident, the `apache.org` server's web root and FTP server area overlapped.
- Furthermore, the FTP area allowed files to be uploaded with insecure permissions.
- The attackers were able to upload a PHP document that allowed them to execute arbitrary commands via the web server.
- The attackers also uploaded a network listener daemon, which they triggered via their PHP document that gave them a shell (as an unprivileged user) on the box.
- Once on the box, they investigated the system and found that the MySQL server was running as root and was accessible from their account. A username and password for accessing the server were found in one of the CGI scripts on the box.
- They were able to coerce the MySQL server into writing arbitrary files (though it wouldn't overwrite existing files). The attackers put a "create set-UID shell" script into `/root/.tcshrc` and waited for one of the admins to `su` to root. Game over.

I've made a copy of the full description of the incident (from the perpetrators themselves) at

http://deer-run.com/~hal/apache-Y2K-defacement.txt.

Perhaps the most humorous part of the write-up is the comment from the attackers: "We didn't wanted [sic] to use any buffer overflow or some lame exploit." Nice to see they have some professional standards…

The Rest of This Section

- Configuring Apache
- SSL Configuration
- **`mod_security`**

The rest of this section is devoted to looking at the security-related issues around running an Apache web server. This is not a complete course on how to administer Apache; it's just enough to allow you to talk to your server admins and content developers about the "right" way to run things.

Configuring Apache covers the basic Apache server configuration directives that can be used to improve security.

SSL Configuration looks at how to turn on SSL support in Apache and also covers topics like getting server certificates.

Finally, we'll take a quick look at *mod_security*, an Apache add-on module that can be used as a Web Application Firewall (WAF).

Configuring Apache

In this section, we'll look at some basic "best practices" to use when configuring your web server. Many of the topics covered here are also covered in the Apache "Security Tips" document available from *http://httpd.apache.org/docs/2.2/misc/security_tips.html*

All configuration directives discussed in this section are set in the standard `httpd.conf` configuration file used by Apache.

Basic Server Config

```
ServerName www.sysiphus.com
UseCanonicalName on
Listen 0.0.0.0:80          # default

User www                   # create special
Group www                  # ditto

ServerAdmin webmaster@sysiphus.com
ServerSignature off
ServerTokens Prod          # hides version
```


Setting the web server's internal `ServerName` may not seem important from a security perspective, but using a single canonical name can be important with sites that are password protected. Basically, this is because browsers cache authentication information by hostname (and one other piece of information, the authentication "realm name", which we will discuss when we talk about password authentication in Apache)—if the web server doesn't use a consistent canonical name, then you may end up prompting users several times for password access to the same information. The `UseCanonicalName` directive tells the web server to always use the value of `ServerName` when constructing so-called "self-referential" URLs so that naming will be consistent throughout your website.

We'll talk about the `Listen` directive on the next slide. `User` and `Group` determine what privileges the web server should run with. Create a specific UID and GID for your web server (don't use `nobody` because other services use that UID) and *never* run web servers as root.

`ServerAdmin` is the email address of the person who should be contacted in case of problems—this address appears in various error messages by default. `ServerSignature` controls whether or not a footer message (which happens to contain the Apache server version number) should be appended to server-generated error messages and other server generated docs. Generally, we'd like to hide the server version by turning this off. Similarly, `ServerTokens` controls the format of the server header returned on HTTP requests. The most minimal setting is "`Prod`" which causes on the word "`Apache`" to be displayed (without version number, etc.)—many automated worms trigger on the Apache version number presented here, so you can slow down infection by hiding this information. If you want to hide that you're running "`Apache`", you'll actually have to hack the `httpd.h` file in the source distribution to change the values of the `SERVER_BASE*` variables and recompile. If you're going to this extreme, though, you'll also want to change the default error documents, etc. because the format of these messages gives away the fact that you're using Apache.

Quick Aside: Local Servers

- Developers may need local web servers
- Security admins worry about potential security disaster with that many servers
- Consider having server bind to localhost:

```
ServerName localhost
Listen 127.0.0.1:8080
```

- Now only local developer sees server on their machine

`Listen` sets the IP address and port number the server is listening on. By default, the web server will try to listen on port 80 for all of the machine's IP addresses (represented as the special address `0.0.0.0`), but the admin may choose to bind the server to a specific address if desired. This could allow the admin to bind the server to a special "protected" interface on the system (and this is also used for setting up web servers on "virtual" IP addresses for hosting multiple sites on the same physical machine). But there's also another use for the `Listen` directive.

In web development shops, it's pretty common for every developer to have their own local web server(s) running for development and testing. That's a lot of web servers on a network—any one of which might be running some untested code that contains a security vulnerability. Not to mention the fact that the web server software itself might have a buffer overflow or other remote exploit. While the network used by your developers is probably protected by your corporate firewalls and security infrastructure, local security admins may worry about having so many essentially "self-supported" web servers running out there.

One option is to use the `Listen` directive to bind the server only to the "loopback" address on the system—address `127.0.0.1`. The developer using the system can still "see" their local web server by pointing their browser at "`http://localhost/`", but nobody outside of the system will be able to access the web server. This seems like a reasonable compromise for everybody concerned.

Start at the Top

- Default policy should block all access:

```
<Directory "/">
    Options None
    Order allow,deny
    Deny from all
    Satisfy all
    AllowOverride None
</Directory>
```

- Override settings for specific dirs as needed

The next step in server configuration is to define access controls for the various directories served by the Apache software. "Best practice" is to first define a general "default deny" stance for the entire file system as we do here. Later, we will configure access to specific directories that are actually part of the web server's document trees.

If we first set a default policy of allowing no access and then have to explicitly enable access for web server directories, this can help prevent "accidental" misconfigurations where material ends up getting into the web server tree that doesn't belong there. This "default deny" policy also helps protect the server against attacks (like the recent "directory traversal" attack reported in Aug 2002) which allows external attackers to "escape" from the document trees and walk the file system of the local server.

So here we're setting options for the directory "`/`", or the root of the file system on downward. These options are put into a `<Directory…></Directory>` "container" which specifies the specific physical directory structure the options apply to. The alternative is to use `<Location…></Location>` which specifies a URL path rather than a physical directory. Frankly, `<Location…>` is confusing—best to stick with the literal path names on your system and use `<Directory…>`.

The next 15 slides or so cover what all of these configuration options mean in great detail …

Setting `Options` Field

- Options can be any/all of the following:
  `Indexes`
  `FollowSymlinks/SymlinksIfOwnerMatch`
  `Includes/IncludesNOEXEC`
  `ExecCGI`
  `MultiViews`
- "`All`" and "`None`" are also supported
- Some options have surprising results …

`Options` defines a list of the optional behaviors that are defined for the web server on a given directory structure. Generally, the `Options` directive is written as a space-separated list of the options you want turned on:

```
Options Indexes Multiviews FollowSymlinks
```

Any option not specifically listed is disabled.

However, an alternate form is to list options with +/- to specifically enable/disable certain options:

```
Options +Indexes +Multiviews +FollowSymlinks -Includes -ExecCGI
```

It turns out the second form shown above can have some unexpected consequences when `Options` is set on both a parent directory and one of its subdirectories—when +/- is used for all options, then the *union* of the `Options` for parent and subdirectory is used. Best to use the first form.

"`Options All`" sets all options *except* `MultiViews`, which must be set explicitly (this appears to be due to the "organic" nature of the growth of Apache—`MultiViews` is a relatively recent development). "`Options None`" disables all options (per our "default deny" stance on the previous slide).

Of all of the options shown here, only `MultiViews` has no real security implications (`MultiViews` enables "server negotiated" content—typically used to select multiple versions of the same web document translated into different languages). But let's take a look at the other options right now…

`Indexes` Option

- Accessing a web directory usually shows `index.html` file
- If no `index.html`, server returns "404 Not Found"
- If `Indexes` is turned on, directory listing is displayed
- May not want this:
  - Can show "private" files
  - Give away docroot structure

Normally, when you view a URL like *`http://www.sans.org/info/`*, the web server displays a file called "`index.html`" in the "`info`" subdirectory of the web server document root (actually, the filename doesn't have to be `index.html`—the filename can be set by the server administrator). If this file is not found, then by default the web server will display the ever-popular "`404 Not Found`" error.

However, if `Indexes` is turned on for a directory, the web server will instead show a directory listing of the other files and subdirectories in the given directory. Sometimes this is useful—particularly if you're using your web server as a mechanism for sharing files. However, it can also be a problem—if somebody deletes the `index.html` file or forgets to create it in the first place, then you may be exposing files and directories that you didn't want to expose. It may also allow remote users to navigate into sections of your document tree that they shouldn't know about (though honestly if the information is that sensitive, it either shouldn't be on the web server in the first place or at least should be password protected).

Generally, it's best to disable `Indexes` unless you're sure you're going to need it. This has the best chance of preventing "accidental" errors that expose your doc root to scrutiny.

Symlink Options

`FollowSymlinks`: Web server will follow symlinks—even out of normal doc tree
`SymlinksIfOwnerMatch`: Link owner and file owner must match

- Disable `FollowSymlinks` to improve security, but has performance impact
- `SymlinksIfOwnerMatch` doubles performance impact

When `FollowSymlinks` is enabled, then the web server will simply "open" any symbolic links in the doc root without question and display the contents of the file pointed to by the symbolic link. This means that content maintainers could accidentally create a symlink that points to a critical system configuration file and expose the contents of that file to the world. Now, of course, this file must still be readable by whatever user the web server is running as, but this still may be something of a concern.

So, the first impulse is to turn `FollowSymlinks` off. However, doing this has an unexpected performance impact. The server must now check each "file" in the doc root before opening it, just to verify that the file is not, in fact, a symlink. This means an extra system call before each and every file opened by the web server—which amounts to a lot of extra work on a busy server. For this reason, many sites choose to leave `FollowSymlinks` enabled. You can scan your document trees periodically for symlinks (perhaps even verifying that the links don't point out of the doc tree), or even run the web server `chroot()`ed if you're really concerned.

Another option is `SymlinksIfOwnerMatch`, which means only open the file pointed to by the symlink if the owner of the file and the owner of the symlink are the same UID. This was designed to prevent web developers from accidentally pointing to root-owned system configuration files. Of course, this now means that the web server potentially has to do *multiple* system calls on each file open—one to check for a symlink, and then additional calls to get the name of the file the link points to and the owner of that file.

Still, if your web server is not heavily loaded, just don't enable `FollowSymlinks` at all.

`Includes` Option

- Allows "server-parsed HTML" capability
- Has some real advantages:
  - Include standard headers/footers
  - Add limited dynamic content w/o using CGI
- Beware `#exec` option, however
- Consider using `IncludesNOEXEC`, which disables `#exec`

`Includes` turns on "server parsed HTML" capability (usually indicated by files with the "`.shtml`" extension). This is a simple way to get bits of dynamic content into otherwise static HTML files without going to the trouble of writing a full CGI or PHP program. Another use for server-parsed HTML is to automatically include certain standard header/footer information in many different documents. For example, some sites include a copyright statement at the bottom of each page—updating the copyright date in a single file is much easier than having to modify all of the web pages on your site.

However, server-parsed HTML includes the `#exec` command, which runs a specified command and substitutes the output of that command into the document (like the `date` command to display the current date/time on the page). This opens the door for a badly written document to execute a command that has unexpected consequences on the system. As an alternative, you can set `IncludesNOEXEC`, which enables server-parsed HTML, but disables `#exec`.

As an aside, we mentioned that server-parsed HTML files are generally distinguished by the "`.shtml`" extension. This is accomplished by setting up a special "handler" for files with this extension:

```
AddType text/html .shtml
AddHandler server-parsed .shtml
```

Another option, however, is to set "`XBitHack on`", which causes any HTML file that has the execute bit set to be treated as a server-parsed HTML file. On the plus side, this means that outsiders won't know that the file is server-parsed HTML because you no longer need the "`.shtml`" extension. On the minus side, it's easy to mess things up and forget to set the execute bit (or lose the bit when copying files from one directory to another) or to set this bit on files that shouldn't have it.

Enabling CGI Access

- Avoid using "`ExecCGI`" which scatters CGI scripts throughout doc trees
- Restrict CGIs to a tightly controlled area:

```
ScriptAlias /cgi-bin/ "/var/apache/cgi-bin/"
<Directory "/var/apache/cgi-bin/">
        Options None
        AllowOverride None
        Order deny,allow
</Directory>
```


There are two ways to enable CGI scripts with Apache—a "right way" and the "wrong way." The "right way" is to restrict CGIs to one (or at worst a small number) of tightly controlled directories that are not part of the normal document tree. Only a few people should be authorized to add CGIs to this directory, and then only after close scrutiny of the program and some serious testing. CGI directories are created with the `ScriptAlias` directive—in the example on the slide, we're saying that `http://www.ourdomain.com/cgi-bin/foo` means "run the program `foo` from the directory `/var/apache/cgi-bin`". Our "default deny" stance that we set up on the root of the file system means that we need to explicitly allow access to this CGI bin directory using the configuration commands in the `<Directory…></Directory>` container.

The alternative to setting up restricted CGI bin directories with `ScriptAlias` is to use the `ExecCGI` option on directories in the document root. Typically, `ExecCGI` needs to be combined with an `AddHandler` directive so that the server can recognize CGI programs:

```
AddHandler cgi-script cgi pl sh
```

Now, any file that ends with the extension `.cgi`, `.pl`, or `.sh` will be executed as a CGI script.

The problem with `ExecCGI` is that now anybody with access to the doc root can drop in a CGI script that could contain a vulnerability that compromises the system. If there are a lot of people providing content to the web server, the situation can get out of control quickly. The Apache "Security Tips" document recommends using `ExecCGI` if and only if:

- You trust your users not to write scripts that will deliberately or accidentally expose your system to an attack.
- You consider security at your site to be so feeble in other areas, as to make one more potential hole irrelevant.
- You have no users, and nobody ever visits your server.

IP-Based Access Control

- Use "`Allow from`" and "`Deny from`" ACLs with:
  - *Hosts:* Use host name or IP address
  - *Networks:* Use "net/mask" or CIDR notation
  - *Domains:* Use domain name
- Must be used with `Order` directive–the confusing part …

Now let's look at how to control access to web directories based on the IP address of the remote machine.

Apache uses "`Allow from`" and "`Deny from`" to permit/deny access based on IP address. You can specify host names or IP addresses of individual machines, or even entire networks using either network/mask syntax (e.g. `192.168.1.0/255.255.255.0`) or CIDR notation (`192.168.1.0/24`). You can even specify domain names. Note, however, that using host names and domain names depends on attackers not being able to spoof or corrupt the local DNS server—best to use IP addresses whenever possible.

By the way, you can also use the special keyword "`all`" ("`Deny from all`", "`Allow from all`") as a wildcard.

If "`Allow from`" and "`Deny from`" were the entire story, then things would be simple. However, there's an extra complication because of Apache's `Order` directive—more on this on the next slide…

The `Order` Directive

- Just remember that `Order` controls:
  - Eval order for `Allow/Deny` commands
  - Default behavior if no matching `Allow/Deny`

"`Order Allow,Deny`"
- Process `Allow` statements, then `Deny` rules
- Default is *access denied*

"`Order Deny,Allow`"
- Do `Deny` statements first, then `Allow` rules
- Access is *allowed* if no match

What's confusing about the `Order` directive is that it actually controls two essentially unrelated parameters. First, the `Order` directive controls the order in which `Allow/Deny` statements are processed—which is sort of intuitive given the name "`Order.`" However, `Order` also defines a *default* policy for connections that don't match one of the explicit `Allow/Deny` statements given by the administrator—and, of course, the default policy is different depending on how the `Order` statement is written.

Basically, the administrator has two options. "`Order Allow,Deny`" says to process the "`Allow from`" statements first, and then all of the "`Deny from`" statements. However, "`Order Allow,Deny`" also means that the default is to drop any connections that don't have an explicit matching "`Allow from`" (in other words, "`Order Allow,Deny`" is the standard *default deny* stance).

The alternative is "`Order Deny,Allow`". Here, the "`Deny from`" statements are processed first, and then the "`Allow from`" statements. The default for "`Order Deny,Allow`" is to permit any connections that aren't explicitly blocked by "`Deny from`" (making this the *default permit* stance).

Confused yet? Let's try the example on the next slide…

An Example

- Suppose we had the following

```
Allow from sysiphus.com
Deny from foo.sysiphus.com
Order Allow,Deny
```

- What happens with various connections?
- Now reverse the `Order` statement—what happens?

Given the example shown on the slide, let's consider a few different scenarios:

- If a connection comes in from the host `foo.sysiphus.com`, it will be denied—first the "`Allow from`" is consulted (access would be permitted because `foo.sysiphus.com` is part of `sysiphus.com`), and then the "`Deny from`" (which explicitly blocks `foo.sysiphus.com`).
- Connections from any other host in `sysiphus.com` would be allowed—these would match the first "`Allow from`" and be allowed, falling through the "`Deny from`" directive because they don't match.
- Connections from all other hosts would be blocked—these connections don't match either the "`Allow from`" or the "`Deny from`", so the implicit "deny everything" caused by "`Order Allow,Deny`" takes over and drops these connections.

OK, that was easy! Now suppose we changed the `Order` line above to "`Order Deny,Allow`" and left everything else the same:

- Now all of a sudden, connections from `foo.sysiphus.com` are *allowed*—the "`Deny from`" is now consulted *first* (which would normally block the connection), but then the "`Allow from`" is checked (which allows connections from anything in `sysiphus.com`—including `foo.sysiphus.com` which otherwise would be blocked).
- Anything else in `sysiphus.com` is also allowed because of the "`Allow from`".
- Other hosts are also *allowed*—remember that our default policy is now "permit everything" because we now have "`Order Deny,Allow`".

Frankly, it's hard to imagine how the Apache folks could have messed this up any worse. They should have simply emulated the same sort of ACLs used by TCP Wrappers and have been done with it, but they didn't.

Granting Access to Doc Root

```
DocumentRoot "/var/apache/docs"

<Directory "/var/apache/docs">
    Options Indexes SymLinksIfOwnerMatch \
            IncludesNOEXEC MultiViews
    Order deny,allow
    AllowOverride None
</Directory>
```


OK, with all of that out of the way, let's look at how we might allow access to our web server document tree. Remember that we implemented a "default deny" stance on the root file system, so we have to explicitly enable access to our `DocumentRoot`.

We do this by setting options on the appropriate directory in another `<Directory…></Directory>` container—these settings will override the settings for the "`/`" directory we set up earlier. First, we set our `Options` directive to include the options we want on our doc root. Next, we need to allow access from all hosts on the internet—setting "`Order deny,allow`" means that our default access control policy is to allow all access.

The `AllowOverride` option will be discussed later in this section.

Password Access

- Directories can be password protected
- Web is stateless—username/password resent for each page
- Standard is "Basic Auth"—passwords are sent in the clear
- Alternative is "Digest Auth"—but there are issues

In addition to IP-address-based access controls, portions of your web space can also be password protected. When entering a protected document space for the first time, the user is prompted for a username/password (this is generally referred to as "HTTP Basic Auth"). What the user doesn't see is that this same username/password is transmitted over and over again on each and every request for a document from that section of the web tree—the browser caches the username/password and simply resends it without prompting the user (when the browser is killed and restarted, the user must re-enter the password). The bad news is that the username/password travels over the network in the clear and can be easily sniffed. So, you should only use HTTP Basic Auth in combination with SSL encryption.

The alternative to Basic Auth is "Digest Auth"—Digest Auth transmits an MD5 hash of the user's password (plus other data) rather than the password itself. This prevents the password from being sniffed directly, but brute-force attacks are possible to recover the password from the MD5 hash. However, Digest Auth generally requires that passwords are stored in cleartext on the server side, which is a problem if your web server is ever compromised. There has also historically been a problem with general browser support for Digest Auth, but all of the major browsers should support it at this point. For more details on issues around Digest Auth, see:

http://en.wikipedia.org/wiki/Digest_access_authentication#HTTP_digest_authentication_considerations

To enable Digest Auth in Apache, you must configure the server with `--enable-digest` and recompile. Further information on configuring Digest Auth can be found at:

http://httpd.apache.org/docs/current/mod/mod_auth_digest.html

Managing Password Files

- Format of Apache password files is:

```
username:password[:ignored]
```

- Don't use `/etc/shadow` file!
- Use `htpasswd` command to manage Apache password file
- Keep password files out of doc trees!!!

To password protect part of the web tree, we first need a password file. The format of Apache password files is simple—just the username, a colon, and an encrypted password string. Anything after the encrypted password is ignored by Apache—so it's possible to simply use a copy of the system `/etc/shadow` file (you'd have to use a copy because `/etc/shadow` wouldn't normally be readable by the web server user). This is a terrible idea for two reasons: (1) you'd have a copy of `/etc/shadow` on your system that was readable by a non-root user, and (2) users should use a different password for web access and login access because Basic Auth insists on retransmitting the username/password all the time. It's also important to put the Apache password files someplace *outside* of the web document tree (nothing is worse than finding your web password file indexed on an internet search engine). As we'll see, Apache allows the administrator to specify the location of the password file anywhere in the file system.

The `htpasswd` command provided with the Apache distribution allows the admin to add/remove users and change passwords. The syntax is "`htpasswd [-ms] <pwdfile> <username>`" where `<pwdfile>` is the path to the Apache password file, and `<username>` is the name of the user being added/changed. By default, `htpasswd` uses old-style DES56 password encryption. The `-m` option tells `htpasswd` to use MD5 encryption, and `-s` uses SHA-1 (the same password file may have different passwords using all three types of encryption).

So, here's how to add user "`hal`" to `/var/apache/etc/passwd` with MD5 password encryption (the same command is also used to change the password for user "`hal`"):

```
htpasswd -m /var/apache/etc/passwd hal
```

In this mode, the admin will be prompted for the password for the user's new password. `htpasswd` also supports "batch mode" where passwords can be entered on the command line (good for scripts):

```
htpasswd -mb /var/apache/etc/passwd hal NewP8ssw0rd
```

Of course, this means that the user's password would be at least momentarily visible via `ps`.

Sample Directory Config

```
<Directory "/var/apache/docs/private">
    AuthType Basic
    AuthName "Private Info"
    AuthUserFile "/var/apache/etc/passwd"
    Require valid-user
</Directory>
```

- "`Require`" directive is *critical*
- Can optionally use Berkeley DB files for performance

Here's how to actually configure password-based access control on part of the doc tree. "`AuthType Basic`" says that we should use HTTP Basic Auth (as opposed to "`AuthType Digest`" for Digest Auth).

"`AuthName`" is a string that is used as part of the username/password prompt by the browser—in this example, the prompt on the pop-up box would read something like "*Enter username/password for Private Info.*" Actually, recalling our earlier discussion of `ServerName` and `UseCanonicalName`, `AuthName` is really the authentication "realm name" that the client browser uses (along with the server name) to remember the username/password credential for the protected area of the remote website. In other words, two password-protected directories on the same web server that used two different values for `AuthName` would require the user to enter a username and a password for each specific directory. If the two directories used the same value for `AuthName`, then the user would only be prompted once.

"`AuthUserFile`" is the location of the Apache password file—again, put this file someplace *outside* your document tree. Different directories can use different password files or all share the same file. Be careful here—if you configure multiple directories with the same `AuthName`, then also configure them to use the same `AuthUserFile`. Otherwise, you could run into problems when the files get out of sync and the user's password that's valid in one file is not valid in the other.

The `Require` directive says that the user must enter a valid username/password before being given access. If you forget to put the `Require` directive into the config file, then the username/password box will pop up but *any username and password will allow access!* The reason the `Require` directive exists at all is that it's also possible to say "`Require hal sally bob`", which would only grant access if the username were one of the three listed users (and obviously only if the correct password for the given account was entered as well).

One problem with standard flat text password files is that as the number of users grows, finding a particular user in the file takes longer and longer. And again, remember that the username/password verification has to be done on each and every document access from the restricted area. For performance reasons, it's possible to create your password files as DBM or Berkeley DB style database files to enable faster lookups.

You must reconfigure Apache with `--enable-auth_db` or `--enable-auth_dbm` flags and recompile before this is possible, however. Once the server has been properly built, simply use `AuthDBMUserFile` instead of `AuthUserFile`. The Apache distribution comes with the `dbmmanage` program for managing user accounts in DB or DBM style database files.

Again, for more info see:

http://httpd.apache.org/docs/current/mod/mod_authn_dbm.html

Access by Password *or* Address

```
<Directory "/var/apache/docs/locals-only">
    AuthType Basic
    AuthName "Local Info"
    AuthUserFile "/var/apache/etc/passwd"
    Require valid-user
    Order allow,deny
    Allow from 192.168.0.0/16
    Allow from 127.0.0.1
    Satisfy any
</Directory>
```


Now it's possible to discuss the "`Satisfy`" directive we saw in an earlier example. `Satisfy` simply controls the relationship between the various forms of access controls defined in a particular part of the doc tree. This includes IP-based access controls and password-based access controls. Basically, the choice is either "`Satisfy all`", which means that both the IP-based restrictions must be satisfied and a proper username/password entered, or "`Satisfy any`" which means that either the IP address must match or a correct username/password must be used.

This latter case is useful when you want to have a section of your web tree that's only visible to "local" people. In the example on the slide, our IP-based access controls allow access from the `192.168.0.0` network—since "`Satisfy any`" is defined, then as long as the connection comes from a host on this network, a password will not be required (in fact, the prompt will never even appear since the IP address of the source of the connection is checked first). Connections from other IP addresses will cause the password pop-up box to appear and force the user to enter a valid password.

What's `AllowOverride` Do?

- `AllowOverride` allows configuration in doc trees

PROS:

- Delegate administration to various groups
- Reconfigure server without restarting

CONS:

- Security policy spread throughout doc tree
- Performance hit

The last configuration command to discuss is the `AllowOverride` option. Enabling `AllowOverride` allows the owner of the document tree to create a configuration file in the doc tree itself, which overrides settings in the `httpd.conf` file (usually these files are named `.htaccess`, but this is customizable as we'll see).

On the plus side, this allows the web server administrator to delegate administration to the owner of the document tree. For large web environments, this may be the only manageable solution. Another advantage is that these configuration files are read every time a document is accessed from the given document tree, so any changes made to the configuration in these files take effect immediately without having to restart the web server (changes to `httpd.conf` require at least a `HUP` to the running daemon).

This is actually also a problem as well. Since the web server has to look for these per-directory configuration files when `AllowOverride` is set, and since these files then need to be parsed by the web server when found, this causes a performance hit for each and every file access. In fact, if `AllowOverride` were set on `/var/apache/docs`, and the file being accessed were in `/var/apache/docs/local/config`, the web server would actually have to look for override files at each level of the tree from `/var/apache/docs` on downward. This is a real problem for deeply nested trees.

The other problem is that the security policy for the web server is now spread throughout the document tree. This not only makes it difficult to know how the server or a particular part of the document tree is configured, it also makes it easier for somebody to make a mistake, which opens up a vulnerability on the machine. And since the mistaken configuration directive is hidden away in some random file in the doc tree, it could go unfound for a long time.

AllowOverride Syntax

AllowOverride can be "All" or "None", or list from:

- Options: Change Options settings
- Limit: Address-based access controls
- AuthConfig: Password auth configuration
- Indexes: Directory listing "look and feel"
- FileInfo: Document and language control

To allow setting the Indexes option, use the Options override, not Indexes!

The web server administrator actually has some level of control over what configuration settings may be overridden in the per-directory config files. AllowOverride can be set to "None", which disables overrides entirely, or "All", which basically means that any configuration option can be overridden. However, the administrator may also choose to only allow overrides for certain groups of commands, as defined by the five options shown here.

"AllowOverride Options" means that the per-directory config files can override any of the Options set in httpd.conf for that directory (it also allows the XBitHack option for server parsed HTML to be turned on here as well). The Limit override allows the per-directory config file to set "Allow from", "Deny from", and "Order" to set IP-based access controls. AuthConfig allows the per-directory config file to set all of the Auth* and Require directives used for password-based access controls.

Indexes and FileInfo don't particularly have any security-related impact. "AllowOverride Indexes" just allows local control over how directory indexes appear but doesn't allow the local config file to change the value of the Indexes option—the web server administrator needs to set "AllowOverride Options" to allow this to happen.

Multiple override options may be specified with a single AllowOverride directive—simply enter a space-separated list of the desired overrides (example on the next slide).

Using `AllowOverride`

```
AccessFileName .htaccess

<FilesMatch "^\.ht">
        Order allow,deny
</FilesMatch>

<Directory "/var/apache/docs/locals-only">
    AllowOverride Limit AuthConfig
</Directory>
```


`AccessFileName` defines the name of the override file recognized by the web server. Again, the default is usually `.htaccess`, but you might want to choose a different standard for your web servers to try and prevent external attackers from stealing these files out of your doc tree.

Whatever name you end up choosing, it's important to prevent these files from being displayed by your web server. Here, we're using the `<FilesMatch...></FilesMatch>` container to deny access to any file on our web server named "`.ht*`". Another example of this useful feature would be to block access to backup files created by common editors (like `*~` files created by Emacs, or `.bak` or `.sav` files):

```
<FilesMatch "*(\~|\.bak|.sav|.orig)$”>
    Order allow,deny
</FilesMatch>
```

Now, we can set `AllowOverride` for our "locals only" document tree that we set up in the previous example. No further configuration would be required in `httpd.conf`—the contents of the `<Directory...></Directory>` container in our previous example would simply go into the `.htaccess` file in the `/var/apache/docs/locals-only` directory.

SSL Configuration

Earlier, we mentioned that it may be useful to combine SSL encryption with HTTP Basic Auth to protect passwords that are being used to access private websites. More commonly, though, SSL is used to protect sensitive data that is flowing between web browser and web server—credit card numbers, personal information, etc. This section looks at how to set up SSL support in Apache and also talks about how to get a server certificate for real internet e-commerce.

Keys and Certificates

- SSL uses public/private key encryption
- Every SSL-ready server needs a key pair
- *Certificate*: Server's public key signed by Certificate Authority
- For e-commerce, cert must be signed by a "recognized" CA

SSL is based on public/private key encryption. Each server that wants to support SSL communications needs a public/private key pair. Browsers expect that the server's public key has been signed by a recognized certificate authority (CA)—this signed public key is referred to as a *server certificate*.

For purely internal use, it's possible to set up your own CA or use dummy certificates that have been signed by a fake CA. This is particularly useful for test environments where you don't want to go to the trouble and expense of acquiring a commercial-grade cert. When a web browser accesses a site that has a dummy certificate, a little pop-up appears warning the user that the certificate is invalid and asking if they want to continue talking to the site. Most users click "yes" and accept the cert without thinking about it.

The reality, though, is that for public e-commerce type activity, you really need to get a site certificate from one of the few CAs that are recognized by the popular web browsers currently in use. This process is described more fully on the next slide…

Getting a Certificate

- Generate key (protect this file!):

```
openssl genrsa [-rand list:of:files] \
    -out server.key 4096
```

- Generate Certificate Signing Request:

```
openssl req -new -key server.key -out server.csr
```

- Transmit CSR (+ money) to CA to receive cert file (`.crt`)

The first step in getting a certificate is to generate a key pair for your web server using the `openssl` command from the OpenSSL distribution. "`openssl genrsa`" generates a standard RSA-type key that is used for standard web browser SSL connections. The `-out` option specifies the file where the key is placed and `4096` is the key length. On systems that do not have a built-in `/dev/random` device, the `openssl` command needs some large amount of apparently random bits to generate the key—the `-rand` option can be used to specify a colon-separated list of files, which can be used to generate pseudo-random values. The usual tactic is to use log files, the system kernel, etc. but not that this approach doesn't really produce truly random values (log files generally use only alphanumeric characters, binaries contain lots of nulls, etc.).

The resulting `server.key` file contains the server's private key, so it's critical that you protect this file. Anybody who steals that key file could impersonate your server and/or act as a man-in-the-middle and compromise SSL sessions between your server and remote browsers. The `openssl` command gives you the option of encrypting this file with a password, but if you do that, then your web server will not be able to start automatically without this password being provided in some fashion (you could code it into the web server boot script, but this doesn't seem much better than not encrypting the file at all). Since the web server starts with root privileges (it needs to bind to port 80 after all), the `server.key` file should be mode 400 owned by root.

Once you've generated the server key, you need to use the `openssl` command again to generate the certificate signing request (CSR) that you will be transmitting to the certificate authority. A list of "recognized" CAs can be found at:

http://httpd.apache.org/docs/2.2/ssl/ssl_faq.html#realcert

There are also cheaper options (like *GoDaddy.com*) that sell what are referred to as *chained certificates.* Basically, these services have their own signing key from the top-level certificate authorities and resell certificates that they've signed with their signing key. Chained certificates "work"—they don't impact the quality of SSL security from your website and users won't receive annoying pop-up warnings about certificates—and they're much cheaper. You will need to enable the `SSLCertificateChainFile` option in your Apache configuration file, however—the instructions you get from the certificate provider will explain how this is done.

Along with the CSR file, you'll also be sending along a quantity of cash money and some proof of the legitimacy of your business. This magic certificate signing rain dance can actually take up to several weeks depending on the CA you choose, so factor that into your timelines. Eventually, you'll get back a signed certificate (`.crt`) file which is what your web server requires (along with the `server.key` file). Note that your certificate needs to be renewed every year or two (more money).

Creating a Self-Signed Cert

Same procedure as requesting a commercial certificate:

- Generate key
- Generate CSR

Just sign the CSR with your own key:

```
openssl x509 -req -days 730 -in server.csr \
    -signkey server.key -out server.crt
```


The first two steps of creating a self-signed cert—creating the key and CSR—are identical to the steps we saw on the previous slide. The only difference is, instead of transmitting your CSR and some money to some external CA, you just use the key you generated to sign the certificate yourself.

If you're going to need multiple dummy certificates for some internal application, you might consider actually creating your own internal CA. That way, all certs can be signed from a consistent root. Furthermore, you could load this internal root certificate into the trusted certificate store on your user desktops, thus eliminating the browser pop-ups about using "untrusted" certificates.

This page has information on installing self-signed certs into the "trusted root certificates" store on Windows machines:

http://blogs.technet.com/sbs/archive/2008/05/08/installing-a-self-signed-certificate-as-a-trusted-root-ca-in-windows-vista.aspx

The next page shows a complete command-line session where I created a self-signed certificate for *www.deer-run.com* ...

```
$ openssl genrsa -out server.key 4096
Generating RSA private key, 4096 bit long modulus
..........+++
.............................................+++
$ openssl req -new -key server.key -out server.csr
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:Oregon
Locality Name (e.g., city) []:Eugene
Organization Name (e.g., company) [WWW Pty Ltd]:Deer Run Associates
Organizational Unit Name (e.g., section) []:
Common Name (e.g., YOUR name) []:www.deer-run.com
Email Address []:webmaster@deer-run.com

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
$ openssl x509 -req -days 730 -in server.csr \
                        -signkey server.key -out server.crt
Signature ok
subject=/C=US/ST=Oregon/L=Eugene/O=Deer Run Associates/CN=www.deer-
run.com/emailAddress=webmaster@deer-run.com
Getting Private key
$ ls -l
total 12
-rw-r--r-- 1 hal hal 1314 2009-08-26 00:12 server.crt
-rw-r--r-- 1 hal hal 1058 2009-08-26 00:11 server.csr
-rw-r--r-- 1 hal hal 1675 2009-08-26 00:09 server.key
```

Configuring SSL

```
Listen 443
          …
<VirtualHost _default_:443>
    ServerName www.sysiphus.com:443
    DocumentRoot "/var/apache/docs-private"
    …
    SSLEngine on
    SSLCertificateFile /var/apache/ssl/server.crt
    SSLCertificateKeyFile /var/apache/ssl/server.key
    SSLCipherSuite …
    …
</VirtualHost>
```


Once you've got your `server.key` file and corresponding `server.crt` file, you need to enable SSL support in `httpd.conf` and tell the web server where these files are installed (again, keep them *out* of the doc trees). The usual configuration (at least for sites with only one web server) is to have an `httpd,` which is listening on port 80 for unencrypted requests and on port 443, which is the standard port for SSL communications (after all, there's no point in shipping around image files via SSL because the encryption adds extra overhead and burns extra CPU cycles on the web server). There's no problem with having multiple `Listen` directives in `httpd.conf`—the web server will bind to all of the requested addresses/ports.

In this configuration, SSL support is generally enabled with a `VirtualHost` definition (`VirtualHost` is also used for supporting multiple different websites via a single Apache instance). Here, we're defining a virtual host, which is listening on port 443 on the default list of addresses that the server is also using for its port 80 bindings. The document root for this virtual server is a different set of documents than those used for unencrypted traffic (it is permissible to use the same document root, but this seems fraught with peril). Note that the security configuration we set up elsewhere (IP-based access controls, password protection, etc.) still applies to the `VirtualHost` directive, so you don't need to repeat all of that again inside of the `VirtualHost` declaration, but you will need to allow access to the new `docs-private` directory since our stance is still "default deny".

We do need to enable SSL (`SSLEngine On`) and tell the server where to find the `server.key` and `server.crt` files. Again, these files—particularly the `server.key` file—should be only readable by root. The files are read by the master Apache process, which runs with root privileges, so restricting access to these files should not be a problem.

There are many, many other SSL options that need to be included, but you can just copy these directly from the sample files provided with the Apache distribution.

Starting Apache with SSL

- Server needs to be started with `-DSSL`:

```
/var/apache/bin/httpd -DSSL \
     -f /var/apache/conf/httpd.conf
```

- "`apachectl startssl`" accomplishes the same thing
- Don't forget to update your boot scripts!

Once you've got `httpd.conf` all set up, you must still start the Apache server with the `-DSSL` command line argument as shown on the slide. You can also use the `apachectl` program—"`apachectl startssl`"—which accomplishes the same thing. Don't forget to update your boot scripts so the web server will be started correctly when the system reboots.

Again, it's sort of a pain that starting up SSL requires a special command-line flag. Why can't the settings in `httpd.conf` be enough? If you don't want to have to bother with the whole "`-DSSL`" nonsense, go into your httpd.conf file and simply remove all of the `<IfDefine SSL>...</IfDefine>` tags. Once these tags are gone, the configuration settings between the tags (i.e., all of the normal SSL startup code) will be executed even if "`-DSSL`" is not set on the command line.

mod_security

`mod_security` is an Open Source Web Application Firewall (WAF) module that can be hooked into Apache. Given that PCI DSS and other security standards are starting to require WAFs, `mod_security` can be a nice way to meet these requirements inexpensively.

We don't have time to go into great detail about the `mod_security` rule language, but I wanted to at least cover the basics of getting `mod_security` installed so that you can start testing its capabilities on your own.

What Is `mod_security`?

Open Source Web Application Firewall:

- Pattern match "signatures" of bad traffic
- Can deny access or just log
- Extensible functionality with Lua scripting

Deployment options:

- Hook directly into Apache instance(s)
- Use in reverse proxy to protect other servers

At its simplest, `mod_security` is a pattern-matching engine that allows you to match "signatures" of various kinds of malicious web traffic. You can also extend these rulesets to do more complicated checks using the Lua scripting language.

`mod_security` is deeply embedded in Apache via the Apache API and can filter not only the incoming HTTP request headers, but can also do deep content inspection (including uploaded file data), and even scan *outgoing* content from the web server to make sure you're not accidentally emitting information you shouldn't like PII and PCI. When you first start using `mod_security`, it's recommended that you run it in "detect only" mode so that it merely logs possible security events—false positives are always a risk. Once you've convinced yourself that `mod_security` isn't going to interfere with your normal web traffic, you can switch over to enforcing mode where `mod_security` blocks access when it detects potentially malicious content.

There are two standard deployment options for `mod_security`:

1. Hook it directly into your web servers that are serving your content. This works fine as long as you have a relatively small number of servers and they're all running some recent version of Apache.
2. Set up an Apache instance with `mod_security` as a reverse proxy between incoming connections from the internet and your web server "farm". This allows one system to protect a large number of web servers, including non-Apache web servers (i.e., IIS).

If you're looking for a commercial solution that utilizes `mod_security`, Trustwave sells COTS appliances based on `mod_security` and hosts the `mod_security` development effort.

What Are the "Core Rules"?

- "Core Rules" block many common web attacks
- Good starting point, good learning tool
- Bundled with `mod_security`
- Start with `DetectionOnly` mode first!

`mod_security` comes with a bundled set of rules called the "Core Rules". This is a fairly extensive set of `mod_security` rules that have been developed over time to detect common sorts of malicious traffic and known exploits. They're also a good overview of the kinds of functionality available in `mod_security`, and serve as an example of how you can write your own rules.

The Core Rules have been developing over many years now, and are quite well tested and tend to have a low false-positive rate. However, remember that when you're first getting started with `mod_security` and the Core Rules, only run in "detect only" mode until you're sure that `mod_security` isn't going to interfere with the normal functioning of your web app.

The Bad News

Not a standard Apache module

May need to build from source

Dependencies:

- XML libs (common on Linux)
- Lua (optional, shared libs required)
- Perl Compatible Reg Ex library (`libpcre`)
- Apache apxs utility (`*-devel` package)

The only rub here is that `mod_security` isn't included with the core Apache distro—generally, you have to download the source code (*http://www.modsecurity.org/download.html*) and compile it yourself.

If you want to include support for the Lua scripting language in your rules, then you also need to download the Lua source (*http://www.lua.org/download.html*) and build it. Note that to use Lua with Apache and `mod_security`, you need to build the embedded Lua interpreter as a shared object, which is not supported by the default Lua build system. Please see the following URL for more information on creating shared objects for Lua:

http://lua-users.org/lists/lua-l/2006-10/msg00091.html

If you need `mod_security` to be able to parse "text/xml" content being handled by your web server, then you will also need the libxml2 library. This library is commonly installed on Linux systems, but you may have to build it yourself on other platforms. Source code is available from *http://xmlsoft.org/downloads.html*

When it's time to build `mod_security`, you'll need a copy of the Perl Compatible Regular Expression library, `libpcre`. This is commonly available as a package in most Linux distros, but make sure you have both the library itself and the `*-devel` package, which contains files that will be used by the `mod_security` build. Similarly, you'll need to install the `*-devel` package associated with your Apache software so that the `mod_security` build will be able to find the `apxs` program required to build Apache modules.

Getting It Working

- Install Core Rules files wherever
- Modify settings in Core Rules config file
- Apache config directives:

```
# Optional functionality
#LoadFile /usr/lib/libxml2.so
#LoadFile /usr/local/lib/liblua.so
LoadModule unique_id_module modules/mod_unique_id.so
LoadModule security2_module modules/mod_security2.so
Include core_rules/*.conf
```


Once you've built and installed the actual `mod_security` module (along with all of its various dependencies), the next step is to install the Core Rules as your initial ruleset. You'll find the Core Rules configuration files in the "`rules`" subdirectory of the `mod_security` source distribution—the `*.conf` files in this directory contain the basic Core Ruleset, and the `*.conf` files in the "`rules/additional_rules`" directory are alternate versions of some of the basic rules and implement a more restrictive ruleset.

Copy the Core Rules `*.conf` files into some directory—I generally install them under my Apache configuration directory in a directory of their own called `core_rules` (`/etc/httpd/core_rules` on most standard Apache installations). You then need to edit the `modsecurity_crs_10_config.conf` file that contains basic settings for `mod_security`. In particular, pay attention to these config settings:

- `SecRuleEngine:` The default setting is "`On`", but you want to set this to "`DetectOnly`" so that `mod_security` is not in enforcing mode initially.
- `SecUploadDir`, `SecDataDir`, `SecTmpDir:` These are set to `/tmp` initially, but I recommend setting these to a directory that only your Apache user has access to. I will typically create `/var/cache/mod_security/{upload,data,tmp}` and make them mode 700 and owned by your web server user.

Be sure to look at the other settings in `modsecurity_crs_10_config.conf` and make sure they're appropriate for your environment and web applications.

The slide shows the configuration directives to add to your `httpd.conf` file (or other configuration directories) in order to enable `mod_security`. You only need to load `libxml2` and `liblua` if you're planning on using that functionality. `mod_security` requires `mod_unique_id`, so make sure to load that module if you haven't already. Finally, we load the `mod_security` module itself and then suck in all of the Core Rules `*.conf` files with an `Include` directive.

Testing

Restart Apache

Enter "<script>" into a web form:

```
[25/May/2019:14:13:08 --0700]
[www.deer-run.com/sid#8e318][rid#90ca8][/][2]
Warning. Pattern match
 "(?:\b(?:(?:type\b\W*?\b(?:text\b\W*?\b..." at
 ARGS:email.
[file "/…core_rules/…40_generic_attacks.conf"]
 [line "102"] [id "950004"] [msg "Cross-site
 Scripting (XSS) Attack"] [data "<script"]
 [severity "CRITICAL"] [tag "WEB_ATTACK/XSS"]
```


Once you've made the appropriate configuration settings, restart Apache. You should immediately see Apache create two new log files in your normal logging directory: `modsec_audit.log` and `modsec_debug.log`. The `modsec_audit.log` file contains full details of the offending requests, and the `modsec_debug.log` file contains differing levels of detail depending on your debug level setting in `modsecurity_crs_10_config.conf`.

Now the question is whether `mod_security` is actually doing something. Surf on over to some web form that's hosted by your web server and enter "`<script>`" in one of the fields. When you submit the form, you should get output like this in your `modsec_debug.log` file (I've artificially introduced some line breaks in the output below for readability; in the normal log file, this would be one big long line):

```
[25/May/2019:14:13:08 --0700]
[www.deer-run.com/sid#8e39198][rid#90cab28][/login/][2]
Warning. Pattern match
"(?:\b(?:(?:type\b\W*?\b(?:text\b\W*?\b(?:j(?:ava)?|ecma|vb)|application\b\
W*?\bx(?:java|vb))script|c(?:opyparentfolder|reatetextrange)|get(?:special|
parent)folder|iframe\b.{0,100}?\bsrc)\b|on(?:(?:mo(?:use(?:o(?:ver|ut)|down
|move|up)|ve)|key(?:press|d ..." at ARGS:email.
[file "/etc/httpd/mod_security/modsecurity_crs_40_generic_attacks.conf"]
[line "102"] [id "950004"] [msg "Cross-site Scripting (XSS) Attack"] [data
"<script"] [severity "CRITICAL"] [tag "WEB_ATTACK/XSS"]
```

As you can see, you get information about the server name and page URL, followed by info about the rule that matched, which `*.conf` file it's in, etc.

Further Reading

- Blog has lots of useful articles
- FAQ and Reference Manual available
- *jwall.org* has policy creation and audit viewer tools
- "Securing WebGoat" definitely pushing the envelope ...

If `mod_security` is interesting to you and you want to learn how to write your own rules, a good place to start is with the Core Rules themselves. The `mod_security` blog postings from Spider Labs (*https://www.trustwave.com/Resources/SpiderLabs-Blog/?tag=ModSecurity/*) have many useful articles about various aspects of rule creation, modification, maintenance, etc. There's also the `mod_security` documentation page (*http://www.modsecurity.org/documentation.html*) which has links to the `mod_security` FAQ and Reference Manual. This page also has links to various `mod_security` documents written by the community.

Some other mod_security-related tools can be found at *jwall.org*. There's a graphical tool for creating mod_security policies and another tool for interpreting the audit logging from mod_security, plus a few other smaller tools.

Google sponsored a project during one of their "Summer of Code" events that uses `mod_security` to mitigate the security vulnerabilities in the OWASP WebGoat application. Considering that WebGoat was an application deliberately *designed* to be insecure (it's a tool for learning about commonly perpetrated web vulnerabilities), actually making it reasonably secure with `mod_security` is something of an achievement. There are lots of crazy `mod_security` rules in this project. More info at:

http://www.owasp.org/index.php/Category:OWASP_Securing_WebGoat_using_ModSecurity_Project

That's All for Now!

- The material presented here is enough to "get you going"
- Really only scratches the surface, though
- Lots of good docs at *www.apache.org*
- Keep up-to-date on new releases!

While this has been a quick introduction to the basic settings for Apache security, there's a lot of additional information available on general web server configuration issues. Most of this documentation is linked off of `www.apache.org`, so take some time to peruse this site. It's also important to check this site regularly to make sure you're up-to-date on the latest Apache releases because most new versions are driven by security-related bug fixes.

Wrap-Up

This page intentionally left blank.

That's It!

- Final Q&A?
- Please Fill Out Your Surveys!

Hal Pomeranz
hal@deer-run.com
http://www.deer-run.com/~hal/
@hal_pomeranz on Twitter

Lab Exercise

- Fun with Apache
- Practice your security configuration chops

Exercises are in HTML files in two places: On the course USB media, and in the home directory of the SANS user in the lab virtual machine. These two locations contain identical copies of the exercise files—it just depends on whether you prefer to look at them from a web browser inside the lab VM (because you're working in full-screen mode) or from your host operating system.

1. Open your web browser
2. Select "File… Open…" from the browser menu (Ctrl-O is the usual keyboard shortcut)
3. This is Day 5, Exercise 3, so navigate to *.../Exercises/Day_5/index.html* and open this HTML file
4. Click on the hyperlink, which is the title of Exercise 3
5. Follow the instructions in the exercise

Appendix A

Sendmail

Some of the material in this section was excerpted from a larger tutorial on administering DNS and Sendmail. If you are interested in more detail on DNS and Sendmail administration (as opposed to security) issues, you can download the PDF for the full course book for this tutorial from:

`http://www.deer-run.com/~hal/dns-sendmail/`

The *DNS and BIND* and *Sendmail* books published by O'Reilly and Associates also make useful references.

Agenda

- Common Security Issues
- Configuring and Running Sendmail
- Running Unprivileged

Common Security Issues is a review of security problems that have historically plagued Sendmail (some of which are appropriate to other Mail Transfer Agents (MTAs) as well).

Configuring and Running Sendmail presents specific examples on how to configure Sendmail to operate securely in the firewall environment we introduced in the *DNS and Bind* portion of this course.

Running Unprivileged discusses where and how to run Sendmail as a non-root user to reduce the impact of future vulnerabilities.

Common Security Issues

Common Security Issues is a review of security problems that have historically plagued Sendmail (some of which are appropriate to other Mail Transfer Agents (MTAs) as well).

Historical Perspective

- Forging email
- Backdoors
- Dangerous aliases
- Addressing attacks
- Privilege Escalation/root Compromise

At a high level, these are the typical problems that have caused heartburn for Sendmail administrators everywhere. Each of these problems is discussed in more detail in upcoming slides. Note that many of these problems are not specific to Sendmail—other MTAs suffer from similar vulnerabilities.

Forging Email

```
% telnet localhost 25
[…]
220 buck.deer-run.com ESMTP Sendmail 8.13.8/8.13.8; …
mail from: santa@northpole.com
250 santa@northpole.com... Sender ok
rcpt to: hal@deer-run.com
250 hal@deer-run.com... Recipient ok
data
354 Enter mail, end with "." on a line by itself
From: Santa@Northpole.com
To: hal@deer-run.com
Subject: Ho

You've been a naughty boy this year!
.
250 KAA06575 Message accepted for delivery
```


The SMTP protocol, unfortunately, makes it very easy to forge email. Simply connect to port 25 on any mail server and compose a bogus message as shown above. Note that the source of the connection will be reported in the mail headers of the resulting message and this information will also go into the mail server's log files, so tracking the source of such forgeries is possible.

The message above is merely humorous. However, there have been several successful attacks where outsiders have forged email from System Administrators in order to dupe users into sending passwords (sometimes credit card information) to an external address. Be sure to explain to your users that you would never send them any email requesting passwords or other private information since, as the administrator, you can change such information at will.

Note that when you connect to a running Sendmail daemon, it reports its version number in the default SMTP greeting. Like BIND, there are attacks that target a specific version of Sendmail, so it is generally a bad idea to advertise this information. As we will see later, Sendmail provides a mechanism for changing this default greeting.

Backdoors

- Sendmail v5 allowed admins to obtain root shell via 25/tcp
- One of the propagation schemes used by the Morris Worm
- The first CERT Advisory (CA-88.01) warns about this

Early versions of Sendmail allowed remote administrators to connect to a running Sendmail daemon and enter "debug" (sometimes "wizard") mode and get a root shell on the remote machine (the internet was a much friendlier place in those days). Since essentially zero authentication was required, this was a huge hole.

As a matter of historical interest, the first CERT Advisory (from 1988!) warns of this vulnerability, though in fact the hole was known for years before the advisory. This hole was one of the primary channels that the Morris Worm used to move around the internet.

Dangerous Aliases

```
# decode: "|/usr/bin/uudecode"
```

Sendmail executes aliases as root

Various aliases can be dangerous

- Overwrite arbitrary files
- Create a set-uid shell
- Give a remote root shell

Sendmail allows administrators to create aliases that pipe mail to programs. Some of these aliases are dangerous and/or the programs themselves are buggy and easily compromised. Since Sendmail executes these aliases with root privilege, very bad things can happen.

The classic example is the `decode` alias (still provided on many operating systems, though usually commented out, thank goodness) which pipes any message sent to this alias to the `uudecode` program. uuencoded files specify the pathname of the file to be unpacked, and an external attacker can use this mechanism to overwrite any system file (like the `passwd` file).

Note that normal users can create `.forward` files that pipe email to programs. Sendmail executes these programs at the privilege level of the given user, but this is still enough privilege for an external attacker to, say, create an `authorized_keys` file in the user's home directory and then log into the machine as that user (possibly later using another attack to get a root shell).

Sendmail ships with a program called `smrsh` (SendMail Restricted SHell) which can be used to safely execute programs in a `chroot()`ed environment. However, we will not cover how to use `smrsh` in this course.

Addressing Attacks

```
'foo@bar.com; rm -rf /*'
```

Requires address be interpreted by a Unix command shell:

- Sometimes triggered through aliases
- Check inputs from web pages!

Another classic Sendmail attack is to send in an email address that contains embedded shell commands. When this email address is handled by a program—for example, used as the reply address on a later invocation of Sendmail—the embedded commands are triggered. The above example is a classic denial-of-service attack, but other attacks can be used to create set-UID shells or modify system files.

The explosion of web-based forms that collect email addresses has produced another avenue for injecting these dangerous email addresses into people's systems. Be sure to carefully check all web inputs before using that information as part of any command!

Another form of addressing attack is to create a message with an extremely long email address in order to trigger a buffer overrun…

root Compromise

Sendmail daemon runs as root

- *Popular "break root" vector because so many systems ran Sendmail by default*

Pre v8.12, `sendmail` binary set-UID to root

- *Popular "privilege escalation" path for attackers*

Again, the fundamental problem here is that Sendmail is running as root. If a remote buffer overflow attack is successful, then an attacker can possibly get a root shell on the remote machine, or at least overwrite a file like the password file to accomplish the same thing. At the end of this section, we'll discuss how to run Sendmail as a non-root user in some cases to help mitigate these issues.

Aside from running as root, `sendmail` was historically installed as a set-UID root binary. There is a class of Sendmail attacks that manipulate the Sendmail binary (via command-line options and sometimes by manipulating the environment) to cause Sendmail to overwrite system files, create set-UID shells, et al. Generally, these attacks allow local, non-privileged users to increase their privilege level. These attacks are often exploited by attackers who have gained unprivileged access to your machine and who want a root shell.

The reason Sendmail has historically been installed set-UID to root is so that the `sendmail` process had the privileges to write outgoing email from users and processes on the system into Sendmail's mail queues. Starting with v8.12.x, Sendmail now has a separate mail queue for outgoing message submission, which is group writable by the special `smmsp` group. So as of v8.12, the `sendmail` binary is normally installed set-GID to `smmsp`, rather than set-UID to root. This is a huge win from a security perspective.

Not only is it critical to stay up to date on Sendmail releases, it is important that you completely eradicate the old binaries when you upgrade. Old set-UID binaries lying around can still be a vector for an attacker.

Other Options

Postfix
http://www.postfix.org/

Qmail
http://cr.yp.to/qmail.html

Exim
http://www.exim.org/

Many attacks against Sendmail have succeeded because it runs as root and is a set-UID root binary. In response to these architectural issues, several other alternate MTAs have sprung up in past years. Of course, Sendmail v8.12.x is now following suit and trying to reduce its use of root privilege.

The major architectural difference between these alternate MTAs and earlier Sendmail versions is that the above mail systems generally break up their functions into multiple separate programs, most of which do not run with superuser privilege. In fact, the only portion of a mail system that generally needs to run as root (and be set-UID to root) is the function that appends email to user mailboxes—this is typically a very simple piece of code and can be exhaustively analyzed for security problems. The process that listens for email, manages the queue, and relays mail to other systems does not need root privilege and can often be `chroot()`ed besides.

There's also, of course, an ongoing religious battle between adherents to the various MTA cults. "My MTA is faster than your MTA!", etc., etc.

Configuring and Running Sendmail

Configuring and Running Sendmail presents specific configuration examples on how to configure Sendmail to operate securely in the firewall environment we introduced in the *DNS and Bind* portion of this course.

The 99.9% Case

➢ *Most hosts don't need a mailer daemon*

- Sendmail daemon does two things:
  - Listens for *incoming* email
  - Process the queue of unsent messages
- If you're not a mail server, then no incoming email
- Queued messages can be handled via safer config

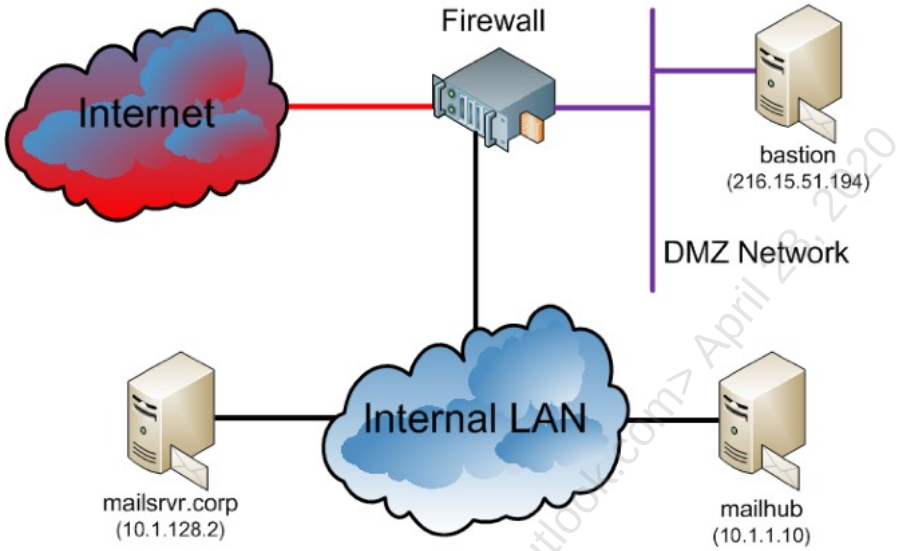
Recall our discussion from earlier in the curriculum. The primary purpose of the Sendmail daemon (or whichever mail daemon you prefer) is to listen on 25/tcp for incoming email from outside the machine. Programs on the system that are sending email out of the machine normally invoke the `sendmail` binary directly from the disk, rather than communicating with the running Sendmail daemon on the system. So, if the machine isn't acting as a mail server, then you should shut off the daemon so that it's not reachable by other hosts on the network—this will protect you the next time a remote exploit is found in Sendmail. This is probably the biggest Sendmail security win because it allows you to focus your attention on the really important machines in your email architecture—the servers—and not worry as much about what's happening on the "client" machines.

See our discussion from earlier in the curriculum on how to manage the outgoing mail queues from your email "client" machines, or read the following articles:

http://www.deer-run.com/~hal/sysadmin/sendmail.html

http://www.deer-run.com/~hal/sysadmin/sendmail2.html

Our Firewall Architecture



Now let's talk about how to create working Sendmail configuration for mail servers. Here's the firewall architecture picture we introduced in the *DNS and BIND* section. Note that we have added another machine to the picture, which is the mail server for one of our theoretical internal-only subdomains.

Mail Routing: Goals

Inbound mail:

- Goes to `bastion` first
- Is immediately forwarded to `mailhub`
- No local delivery on `bastion`

Outbound mail:

- Relayed from internal hosts to `bastion`
- `bastion` delivers to remote domain

The external mail gateway will be used as a mail proxy for both inbound and outbound mail. Mail coming in from outside your organization has to stop on the `bastion` host and then be forwarded into your internal network. No local delivery will ever take place on the `bastion` host itself, thwarting many popular Sendmail-based attacks.

Since we are assuming that no internal hosts can directly reach hosts on the internet, all outbound mail has to find its way to your `bastion` host, which in turn will deliver the mail to its final destination.

Given Sendmail's history of security problems, you may feel the risk of running Sendmail on your external mail server is too great. You may wish to investigate running an alternate MTA (Qmail, Postfix, Exim), at least on your external relay. On the other hand, the Sendmail sources have probably received much more scrutiny than the Qmail or Postfix sources, and perhaps may actually be more secure than the newer alternatives. Also, later in this module, I'll show you how to run Sendmail as a non-root user on your external mail relay servers, which further helps reduce the impact of future Sendmail vulnerabilities.

Creating Sendmail Configs

As of v8, Sendmail config files built from macro definitions

Use "`m4`" to create actual config files:

```
m4 myfile.mc > myfile.cf
cp myfile.cf /etc/mail/sendmail.cf
```

Recommend collecting macro files together in one place

The rest of this section deals primarily with creating appropriate Sendmail configuration files for machines in our example network architecture. In the bad old days, creating Sendmail configuration files required learning a cryptic configuration language. Sendmail v8 made matters easier by defining a set of macros that allow administrators to easily define their Sendmail configurations with a set of (reasonably) English-like declarations.

So, first, the administrator creates a file containing the macro settings that correspond to their desired Sendmail configuration ("`myfile.mc`" in the example on the slide)—we'll see plenty of examples of these files in the upcoming slides. These macros are processed and expanded by the "`m4`" program (a very old program from the early days of Unix) into the standard Sendmail configuration language. Note that `m4` simply spits its output to the standard output, so this output must be redirected into a file. The resulting file should then be copied into place as `/etc/mail/sendmail.cf`, which is Sendmail's default configuration file location.

The standard Sendmail macro definitions can be found in the "`cf`" subdirectory of the Sendmail distribution. For any reasonably large site, you will probably end up creating lots of custom Sendmail configuration files. I find that it's usually a good idea to collect all of these different configuration files into a single location, and keeping them in a subdirectory of the "`cf`" directory seems to work well. Typically, I'll name this directory for the site's local domain name ("`cf/sysiphus.com`" for example) and name the individual files in this directory for the machine they apply to ("`bastion.mc`", "`bastion.cf`", etc.).

Config for bastion

```
include(`../m4/cf.m4')
OSTYPE(`linux')

define(`confSMTP_LOGIN_MSG', `$j mailer ready at $b')
define(`confPRIVACY_FLAGS',
       ``noexpn,novrfy,noverb,noetrn'')
define(`LOCAL_SHELL_PATH', `/dev/null')
define(`confMAX_HOP',`50')

MASQUERADE_AS(sysiphus.com)
FEATURE(`mailertable', `hash -o /etc/mail/mailertable')
define(`MAIL_HUB', `mailhub.sysiphus.com')
MAILER(smtp)
```


When you `telnet` to port 25 on a machine running Sendmail, you see a greeting string that includes the version of Sendmail. Advertising which version of Sendmail you are running is a bad idea, since it can help attackers target your system with specific exploits. The new `SMTP_LOGIN_MSG` we set looks like

```
220 bastion.sysiphus.com ESMTP mailer ready at <date>
```

which doesn't even advertise that we're running Sendmail.

The `PRIVACY_FLAGS` setting turns off the `EXPN` and `VRFY` commands and prevents outsiders from gaining information about users in your domain. We're also disabling the VERB command, which can allow attackers to gather information about your internal mail architecture. ETRN allows remote users to request that your mail server run its queue looking for any queued mail for the remote site. While this is useful in an ISP environment, it could also be used as a potential denial-of-service, so we disable it here.

Changing `LOCAL_SHELL_PATH` to `/dev/null` prevents the local Sendmail daemon from executing any aliases that pipe mail to a program. This should never be a factor anyway, since all locally delivered mail should be forwarded to `mailhub` by the MAIL_HUB directive (see below).

`MAX_HOP` is the number of times a given piece of email can be forwarded before Sendmail kicks the email back to sender—this is used to prevent mailing loops, but the default value (18) may be too low if your mail has to jump through a lot of hoops to get through your firewall.

`MAIL_HUB` forces all mail for the local machine to be forwarded to `mailhub` (nobody from the outside world should be sending mail to `bastion` but cron and other system processes may generate email). The `mailertable` feature (which we'll cover in more detail in the next couple of slides) is a way of creating a simple database of special-case email routing rules for the email being sent to users in our local email domains. All other mail that isn't explicitly handled by a rule in our `mailertable` database—i.e., outgoing mail destined for non-local domains—will be delivered normally via the SMTP mailer.

`MASQUERADE_AS` causes all mail originating from this host to appear to be from `user@sysiphus.com` rather than `user@bastion.sysiphus.com`.

/etc/mail/mailertable

```
corp.sysiphus.com       smtp:mailhub.sysiphus.com
eng.sysiphus.com        smtp:mailhub.sysiphus.com
sysiphus.com            smtp:mailhub.sysiphus.com
```


The `mailertable` database is just a simple mapping of email domain names to machines that should receive email for the given domain. First, we create a text file called `/etc/mail/mailertable`. The left-hand column gives a domain name and the right-hand column specifies a mailer to use and a destination host.

In our case, all we're trying to do is fire all email for our local email domains into the `mailhub` machine for further routing, so our rules are all essentially identical. The nice thing about this configuration, however, is that the only internal IP address the `bastion` host needs to know now is the address of the `mailhub` machine. You could simply put this address in the `/etc/hosts` file on the `bastion` host and then this machine wouldn't even have to look at your internal DNS infrastructure.

Although I'm not going to show you the Sendmail configuration for the `mailhub` machine, it's likely that the `mailhub` also uses a `mailertable` to handle routing email within our hypothetical corporate environment. Of course, the rules in the `mailertable` on the `mailhub` machine would look much different from the rules we're using here on the bastion host.

How To Create DB Files

Sendmail sources include `makemap`

```
% cd sendmail-8.x.x/makemap
% sh Build              (lots of output)
% /bin/su
Password:
# cp obj.<arch>/makemap /etc/mail
# cd /etc/mail
# vi mailertable       (per previous slide)
# ./makemap hash mailertable < mailertable
#
```


Our `mailertable` text file must be converted into a Unix binary database file so that Sendmail can use the information. Your operating system probably already includes a program called `makemap`, which performs this conversion. If your OS doesn't come with `makemap`, the Sendmail distribution includes the `makemap` source code. As you can see, building the software is trivial. I usually install `makemap` in `/etc/mail` if it's not already included with my OS.

Using the `makemap` program is also straightforward. First, you specify the kind of database file to create—here, we're using a standard "Berkeley DB" `hash`-style database, which is typical for Linux and other Open Source OSes (older proprietary Unix systems like Solaris and HP-UX may use "`dbm`" instead of "`hash`" here). The second argument is the "base name" of the database file: For a `hash` style database, `makemap` will actually create a file called "`mailertable.db`" (`dbm` style databases will produce two files, `mailertable.dir` and `mailertable.pag`). `makemap` wants to get the contents of our `mailertable` text file on its standard input, so we're just using simple shell redirection to make that happen.

Note that every time you update the text file, you must rerun the `makemap` program to rebuild the database file. You may want to create a `Makefile` or shell script that does this task automatically.

relay-domains File

- Examples of "good" relaying:
  - Inbound email to local users
  - Outgoing email from local users
- Everything else is "bad"
- Sendmail uses `relay-domains` config:

```
sysiphus.com
eng.sysiphus.com
corp.sysiphus.com
216.15.51
10.1
```


While it's the job of the bastion host to relay email into and out of our hypothetical company, we have to be careful not to allow too much. Email from the internet to our local email domains is fine, as is outgoing email from our users to other sites on the internet. However, any other relaying would mean that we were accepting email from outside sites and relaying it to non-local users. This is referred to as being a *promiscuous* or *open relay*. When your mail servers are an open relay, spammers will use them to forward their junk email to other sites. Not only is this a waste of your bandwidth and resources, but it also means that your mail servers end up getting blacklisted by much of the internet with the result that most sites won't even accept legitimate email from you.

Sendmail controls relaying via the `/etc/mail/relay-domains` file. First, you should list all local email domains you want to pass incoming email for—in our case, we want to make sure that `bastion` passes along email for our internal domains. The `relay-domains` file should also contain the IP address ranges of machines that our mail server will relay *outgoing* email for (you don't want to rely on email domains for this because the spammers could just forge the sender email address to relay email through your mail servers). In this case, we're adding our internal address space to the `relay-domains` file so that our internal users can send mail back out to the rest of the internet.

Config for `mailsrvr.corp`

```
include(`../m4/cf.m4')
OSTYPE(`linux')

define(`confMAX_HOP',`50')
define(`confSMTP_LOGIN_MSG', `$j mailer ready at $b')

FEATURE(promiscuous_relay)
FEATURE(accept_unqualified_senders)
FEATURE(use_cw_file)

define(`SMART_HOST', `mailhub.sysiphus.com')
MASQUERADE_AS(corp.sysiphus.com)
MAILER(smtp)
```


Here's an m4 macro file for `mailsrvr.corp.sysiphus.com`, which will demonstrate some of the different Sendmail features that might be appropriate for *internal* mail servers in your environment.

`promiscuous_relay` and `accept_unqualified_senders` turn off some default Sendmail anti-spam checks. While these features should *never* be disabled on your external mail relay because they help prevent the flow of spam, many sites find that disabling these features makes life easier on their internal mail servers. For example, `promiscuous_relay` tells your mail server that it's OK to relay email from one part of your company to another—without having to mess with the `relay-domains` file or other local configuration. Similarly, `accept_unqualified_senders` means that the mail server will accept email from broken mail clients that simply show the sender's address as a username ("`bob`") without a domain qualifier. The `MASQUERADE_AS` directive will make sure that these emails get the correct domain name appended to them before they're sent on to their final destination.

The `use_cw_file` directive tells Sendmail to look in the file `/etc/mail/local-host-names` for a list of domains (one per line) that are to be considered local (prior to Sendmail v8.10, the file was called `/etc/mail/sendmail.cw`, which is how the `FEATURE` gets its name). In this case, the local-host-names file will only contain "corp.sysiphus.com", but this feature is particularly useful if you accept mail for a lot of domains and is generally easier than editing the `sendmail.cf` file to update this information.
`SMART_HOST` says to forward all email that isn't local to this machine to `mailhub` for further processing. This means that `mailsrvr.corp` will send all mail to `mailhub` that isn't specifically destined for `corp.sysiphus.com` (whether that's outgoing email or email for another subdomain like `eng.sysiphus.com`). This means the `corp.sysiphus.com` mail admins can have an easier time because they don't have to configure and maintain all of the different email routes for the company, but the downside is that `mailhub` now becomes a single point of failure. Consider deploying multiple machines in parallel for redundancy.

`SMART_HOST` is essentially the logical inverse of the `MAIL_HUB` directive we used on `bastion`.

Running Unprivileged

Running Unprivileged discusses where and how to run Sendmail as a non-root user to reduce the impact of future vulnerabilities.

Do I Have to Run as `root`?

Why Sendmail runs as **`root`**:

- Bind to port 25/tcp
- Perform "local delivery" for users
- Read and write config files/queues

But ...

- Give up privileges after binding to 25/tcp
- No local delivery on `bastion`
- Change permissions on files and queue

Sendmail runs as root for several reasons:

- Only processes running as root are allowed to bind to network ports below 1024, at least on Unix machines (on Windows systems, for example, any user can bind to any port). Since the MTA daemon needs to bind to port 25/tcp, it needs to be root.
- When Sendmail is performing local delivery, it needs to be root so that it can append email to different user's mailboxes.
- Various Sendmail configuration files and directories, like the aliases database and the mail queue directory, are only accessible to the root user. Sendmail must run as root to access this information.

However, there are workarounds for all of the above issues—at least on the `bastion` host:

- If you look at other servers on Unix that need to bind to low port numbers—BIND, for example—the daemon will run as root only for as long as it needs to bind to the low port number. Thereafter, it will give up root privileges and run as an unprivileged user.
- We are not doing any kind of local delivery on the `bastion`, so appending mail to user mailboxes is not an issue for us.
- While you don't want many of Sendmail's critical files and directories to be readable by normal users on the system, that doesn't mean those files and directories have to be owned by the root user. We can create a special user just for Sendmail, and make everything owned by that user instead.

It's actually pretty straightforward to get all this setup and working on your external relay servers—you could even use this configuration on your internal relay servers as long as you're not doing local delivery there. There's just a little bit of system reconfiguration and then a new configuration directive in your Sendmail macro definition file. Just follow along with me on the next two slides.

System Configuration

Create new **`sendmail`** user/group

Shut down Sendmail daemon

Reset permissions on critical dirs:

```
chown -R sendmail:sendmail /var/spool/mqueue
chgrp -R sendmail /etc/mail
chmod -R g+r /etc/mail
chmod g+s /etc/mail
```


First, create a new user ID and group ID that your Sendmail will run as. *Do not* use the special "`smmsp`" user and group already in use by Sendmail. I will often create a user and group called "`sendmail`" with UID and GID 24 (`smmsp` is usually UID and GID 25):

```
groupadd -g 24 sendmail
useradd -u 24 -g 24 -M -d /var/spool/mqueue \
    -s /dev/null sendmail
```

Now before you start messing around with ownerships and permissions of various files and directories, it's a good idea to shut down the running Sendmail daemon so that it doesn't get confused while you're in the middle of making your changes. There are really two critical directories for the MTA process: The queue directory and the `/etc/mail` configuration area. The queue directory should be owned by whatever user and group you created to run the MTA daemon as ("`sendmail`" and "`sendmail`" in our example). The permissions on this directory don't need to be changed—it's already mode 700 so only root can access the files in there; we're just changing which user is allowed to peek.

You still want the `/etc/mail` directory and the files in it to be owned by root, because you only want legitimate system administrators to be messing around with these files. But we need to make sure the files in this directory are at least readable by the Sendmail daemon when it's running unprivileged. So, what we're going to do is change the group ownership in the directory and its contents to our "`sendmail`" group, and then use "`chmod -R g+r /etc/mail`" to give group read permissions to everything in the directory. Actually, the files in there are probably already group readable, but it never hurts to be sure.

The last "`chmod g+s /etc/mail`" command adds the so-called "set-GID" bit onto the `/etc/mail` directory. On Unix systems, if set-GID is set on a directory, then any new file created in that directory will automatically inherit the group ownership of the directory (as opposed to the group membership of the user that creates the file, which would be the default). In this way, we can make sure that any newly created files end up with the proper ownerships—like when you're rebuilding the `mailertable` database with the `makemap` program.

RUN_AS_USER

```
include(`../m4/cf.m4')
OSTYPE(`linux')
define(`confRUN_AS_USER', `sendmail:sendmail')
define(`confTRUSTED_USERS', `sendmail')
define(`confSMTP_LOGIN_MSG', `$j mailer ready at $b')
define(`confPRIVACY_FLAGS',
       ``noexpn,novrfy,noverb,noetrn'')
define(`LOCAL_SHELL_PATH', `/dev/null')
define(`confMAX_HOP',`50')
MASQUERADE_AS(sysiphus.com)
FEATURE(`mailertable', `hash -o /etc/mail/mailertable')
define(`MAIL_HUB', `mailhub.sysiphus.com')
MAILER(smtp)
```


Sendmail allows you to specify an alternate user and group to run as with the `RUN_AS_USER` configuration option. As you can see, we're specifying the "`sendmail`" user and group we created per the instructions on the previous slide. The `TRUSTED_USERS` setting adds our `sendmail` user to the list of users who have special administrative privileges—this setting eliminates some spurious error messages from your Sendmail logs when the aliases database and other files are owned by the `sendmail` user.

If you look at the running Sendmail process after you set this option, you may be surprised to see that it's still running as root—this is expected behavior. The master process always runs as root, but when a new SMTP connection comes in, the master process must `fork()` a copy of itself to handle the incoming connection. The "child" process will run as the unprivileged user and group you specify with the `RUN_AS_USER` option. So, while the master Sendmail process itself runs as root, outsiders will only ever be able to communicate with unprivileged child processes.

Anyway, once you've run your new macro definitions through `m4`, copy the resulting config file out to your external server(s) and install it as `/etc/mail/sendmail.cf`. Then restart Sendmail. You can test the configuration by using `telnet` to connect to port 25 ("`telnet localhost 25`"). If you get a process listing in another window, you should see a `sendmail` process running as the `sendmail` user. Close the `telnet` session to make this process go away.

Summary

Security thoughts:

- Disable Sendmail daemon on most systems
- Run unprivileged on pure relay servers
- Change greeting string to hide version
- Disable other features where possible

Email routing:

- `MAIL_HUB` for preventing local delivery
- `SMART_HOST` for outbound mail
- `mailertable` for everything else

We've covered a whole bunch of different Sendmail configuration options, so it's probably a good idea to summarize things before leaving this topic. The most important step for Sendmail security is to disable the Sendmail daemon on all of the machines you own that are not acting as mail servers (which should be 99% of the machines in your environment). Machines that are acting purely as email relay servers (no local delivery) can be configured to run Sendmail as an unprivileged user to help mitigate future vulnerabilities. Actually, once we have Sendmail running as an unprivileged user, it should be reasonable to run the daemon `chroot()`ed as well, but that configuration is beyond the scope of this course.

On your external mail servers particularly, it's a good idea to change Sendmail's default greeting string to hide what version of Sendmail you're using. You may also be able to disable other features—like setting `LOCAL_SHELL_PATH` to disable aliases that pipe email to other programs or turning off SMTP commands like `EXPN`, `VRFY`, `VERB`, and `ETRN`.

The trickiest part about modern firewall architectures is getting your email to route properly in and out of your organization. The `mailertable` feature can be used to create special-case routing rules for most circumstances, and the `SMART_HOST` feature can be used by internal servers to forward email to a central machine for proper routing. `MAIL_HUB` is useful for making sure that local delivery never happens on your external mail relay servers.

Virus Protection

- Filtering out viruses and phishing scams also critical
- MIMEDefang can be added into Sendmail via "Milter"
- Plug ClamAV or other antivirus into MIMEDefang
- Can also add SpamAssassin, Razor, etc. for spam control

While not strictly related to the security of Sendmail, protecting your organization from email-borne viruses and phishing scams is obviously important. While configuring this functionality is outside of the scope of this course, a few pointers are in order.

Most of the solutions in this area for Sendmail make use of the so-called "Milter" (a corruption of "mail filter") interface that appeared in Sendmail v8.11. Basically, the Milter spec defines an API for plugging external modules into Sendmail that allow you to filter messages as they're passing through your server.

MIMEDefang is a Milter that really just acts as a framework for other modules. For example, MIMEDefang has hooks to do virus scanning with the Open Source ClamAV scanner on incoming email (MIMEDefang also has hooks for many of the popular commercial antivirus scanners as well). Recent versions of ClamAV will even recognize certain phishing scams in addition to standard email worms and viruses.

In addition, MIMEDefang has hooks for all of the popular spam control solutions like SpamAssassin, Razor, and others. It's a pretty effective and comprehensive solution and relatively easy to install.

The best place to start is the MIMEDefang "how-to" at *http://www.mickeyhill.com/mimedefang-howto/*. Specific instructions for installing ClamAV can be found at *www.clamav.net*. The SpamAssassin project page is *spamassassin.apache.org*.

Appendix B

PortSentry

More details on PortSentry configuration, if the discussion in the SELinux section made you curious.

Simple Model

- PortSentry daemon binds to unused network ports
- Any traffic hitting these ports is logged
- PortSentry daemon can also:
  - "Null route" the source of the packet
  - Put block rules into local firewall config
  - Run an arbitrary command

PortSentry is really a very simpleminded tool (simple solutions are sometimes the best). Basically, you run the PortSentry daemon, which binds to a list of ports that you specify. These should be ports where you don't normally expect to see any traffic. If PortSentry sees traffic on these ports, then it's probably somebody running a port scanner against you or probing around for some backdoor infection on your machine.

PortSentry will log the unexpected access and can optionally take action. Built-in actions include inserting a route into the system's local routing table so that any packets your host might try to send back to the remote machine just go to a non-existent router and never leave the network. PortSentry can also interact with an IP filter or iptables firewall or with TCP Wrappers to block all future IP traffic from the remote system. Since most port scans are precursors to an attack, rapidly responding to the scan and blocking future traffic can be a big win.

PortSentry also allows the administrator to define an arbitrary command that will be run—either in addition to one of the built-in blocking actions or instead of those actions. This could, for example, allow you to update the firewall rules on your network-layer firewall rather than just the local system.

Of course, all of this firewall talk leads to an important point. If your system is already protected by a network-layer or host-based firewall, then you may be already blocking all of the traffic that PortSentry would alarm on. On the other hand, PortSentry can provide a second level of verification that your firewalls are actually working properly, or monitor network traffic on the relatively wide-open networks *behind* your firewalls, or be used on systems that need to be "exposed" to hostile network environments.

Problem with Active Response

- Might block legitimate traffic due to:
  - Scans from spoofed source IPs
  - Accidental probes from the confused
- Tune ports and "trigger sensitivity" via config file
- Docs discuss why this is less likely than you might think

If you configure PortSentry to automatically block traffic after a probe from some remote system, there is a danger that you end up triggering a false alarm and blocking traffic from somebody you actually want to talk to. For example, somebody might try to reach your web server on port 80, but you're only doing HTTPS traffic on port 443. If you've got PortSentry watching port 80, you may end up blocking a lot of people unintentionally. Or an attacker who figures out that you're using PortSentry might scan you with the spoofed IP address of a machine that you really want to talk to (like your log server) in order to mess with you.

Obviously, being sensible about the ports you monitor with PortSentry is critical to reducing your false alarm rate. PortSentry also allows you to wait until you reach a certain number of probes from a system (say 2 or 3) before triggering a response.

The `README.stealth` file in the PortSentry source distribution also makes some good arguments why scans from forged IP addresses are not really likely. Basically, the arguments boil down to:

- Attackers usually want to get the responses to their port scan, so using a forged address doesn't help them.
- Using forged addresses for decoy scans only increases the amount of time required to do the scan as well as increasing the total amount of network traffic, making the scan easier to spot.
- More and more sites are implementing egress filtering, which prevents spoofed packets from even leaving their networks.

Your mileage, as always, may vary. Perhaps you will want to run PortSentry in "log only" mode until you're convinced that your false alarm rate is acceptable.

Another PortSentry Issue

- Monitored ports appear as "live" to **`netstat`**, **`nmap`**, ...
- Can create false positives when security scanners are run
- Large number of open ports may make host a target

When PortSentry is running on your machine, it binds to the list of ports you specify, just like any normal network service. This means the ports in your list show up as "active" network ports in the output of netstat or lsof. And when the machine is scanned from the outside with a port scanning tool like Nmap, the ports show up as "live" in the scan output.

This causes all kinds of consternation when you run host-based or network-based security audits.

"Why are all those ports open on your machine?"
"Well, they're not really open, per se…"
You get the idea.

If the bad guys are scanning you with Nmap and they find a machine with dozens of open ports, which machine do you think they're going to try to attack first? This may become a nuisance.

Linux "Stealth Mode"

- PortSentry on Linux can listen on ports via raw sockets
- Ports now invisible to **`netstat`**, **`nmap`**, ...
- Can detect half-open and other "stealth" type scans
- Not portable to other systems

In an effort to deal with this problem, PortSentry can take advantage of a peculiarity of the Linux raw socket interface. The Linux kernel allows PortSentry to listen on specific network ports via raw sockets, rather than "binding" to ports in a traditional sense. This means that the ports being monitored don't show up as "active" in the output of `netstat`, `lsof`, and `Nmap`.

What's more interesting is that when PortSentry is listening via raw sockets, it can see various "stealth" scanning modes like half-open connection scanning, SYN-FIN scans, etc., that would not normally be visible to normally bound network sockets. Basically, if PortSentry is bound to the socket via the traditional mechanism, it will only see full 3-way handshake type connections on its TCP ports.

Unfortunately, this raw socket approach is not portable to other operating system platforms where raw sockets behave differently than under Linux. A portable approach would be to have PortSentry put the interface into promiscuous mode and use something like `libpcap` to capture traffic. Unfortunately, this would also hurt performance on the system.

Installation Notes

Set in **`portsentry_config.h`**

- Config file location
- TCP Wrappers config location
- Syslog facility/priority
- Number of hosts to "remember"

Also have to set paths in **`Makefile`**

Build with "**`make`** <*target*>"

PortSentry doesn't come with a `configure` script, so certain parameters must be set manually.

The `portsentry_config.h` file contains several configuration settings that get hard-coded into the binary. `CONFIG_FILE` is the path to PortSentry's configuration file, and `WRAPPER_HOSTS_DENY` is the path to the TCP Wrappers `hosts.deny` file on the system. Additionally, you can define what Syslog facility to use and what level to log at—I prefer that PortSentry logs to `LOG_AUTH`, rather than the default, which is `LOG_DAEMON`. You can also set the number of hosts that PortSentry should keep state for (the default is 50, but feel free to set this higher).

Unfortunately, there are also paths in the `Makefile` that need to get set as well, like the installation directory.

Once you're all done, you just run "`make` *<target>*", where *<target>* is something like "`linux`" or "`solaris`". "`make`" with no arguments displays the list of available targets.

Basic Configuration

Pathnames set in **`portsentry.conf`**–

- **`IGNORE_FILE`**
  List of IP addrs to ignore–don't ignore too much!
- **`BLOCKED_FILE`**
  IP addresses that have been blocked during this session
- **`HISTORY_FILE`**
  Permanent running history of blocked hosts

Set **`RESOLVE_HOST="0"`** to turn off reverse DNS queries

Once you've got the binary built, a "`make install`" will copy the binary and some sample config files into the installation directory you specified in the `Makefile`. The default configuration file for PortSentry is the `portsentry.conf` file in this directory. The sample config file that comes with PortSentry is full of comments to help you along. First, let's take a look at some of the basic settings in the file.

`IGNORE_FILE` (usually `$INST/portsentry.ignore`) is a list of IP addresses or networks in CIDR notation that PortSentry should simply ignore. The usual tactic is to list 127.0.0.1, 0.0.0.0, and the IP addresses of the local interfaces on the machine in this file. You might be tempted to list your entire "internal" network range, but this means you won't detect attacks from infected machines on your own networks.

`BLOCKED_FILE` is a temporary file that lists the hosts that have been blocked by the current instance of the PortSentry daemon. This file is started from scratch every time you restart the daemon. On the other hand, `HISTORY_FILE` is the chronological history of the traffic that the PortSentry daemon has detected and responded to.

If you don't trust DNS and/or your machine is seeing a lot of traffic, you may want to set `RESOLVE_HOST="0"` to disable reverse DNS lookups.

Configuring Ports

Set both **`TCP_PORTS`** and **`UDP_PORTS`**

General tips:

- Avoid ports in use by legitimate services
- Monitor low ports to detect scans quicker
- Monitor ports of well-known backdoors
- Monitor ports of disabled services

Port lists provided with PortSentry definitely need tuning!

The next step is to tune your initial lists of ports that PortSentry should be monitoring. There's one configuration variable for TCP ports and another for UDP ports. PortSentry has an internal limit of 64 ports that it is willing to monitor—actually 64 ports each for TCP and UDP. You can increase this value by changing the `MAXSOCKS` line in `portsentry.h`, but 64 ports are probably more than enough for most uses.

The most important factor to consider when setting up your port lists is to avoid ports where you have network services already running or expect to see legitimate network traffic. This will help you avoid false alarms.

Other than that, make sure you monitor several low-numbered ports so you can quickly detect trivial port scans that start at port 1 and work their way up. You also want to have ports scattered throughout the entire 65K port range so that you have good coverage. Adding ports for popular backdoor Trojans (31337, 5554, etc.) is also a good idea. If you've specifically disabled a service because of security concerns (e.g., `telnetd`) then you might want to monitor for people probing for that service.

Note that the port lists in the default `portsentry.conf` file are not well tuned. They contain ports like DHCP (68 UDP), the RPC portmapper (111 TCP and UDP), NFS (2049 TCP and UDP), and X Windows (6000 TCP) that are surely going to trigger false alarms on many systems.

Response Options

Potential probe attempts always logged

Three different response options:

- **`KILL_ROUTE:`** Adjust routing table or firewall rules
- **`KILL_HOSTS_DENY:`** Block via TCP Wrappers
- **`KILL_RUN_CMD:`** Any other arbitrary command line

Any or all of these may be undefined

PortSentry will always log probes on the ports it's monitoring via Syslog. This functionality cannot be shut off.

However, you can control what actions PortSentry will take when it sees a probe. The `KILL_ROUTE` option is usually set to a command that either adjusts the local system's routing table to black hole traffic back to the host sending the probes or which puts a block rule into the local host-based firewall config to block all traffic from the host. Generally, blocking with a host-based firewall is preferred to "null routing" the host because the firewall actually blocks incoming traffic from the host, whereas the routing solution still allows the bad guy to send "packets of death" to your machine.

`KILL_HOSTS_DENY` is the TCP Wrappers syntax you want to go into `hosts.deny` to block the host. Of course, you need to be careful that there's no permit rule in `hosts.allow` that might allow the traffic through (remember `hosts.allow` is checked before `hosts.deny`). Again, using a host-based firewall is more effective.

Finally, `KILL_RUN_CMD` is an arbitrary command line that will be run either before or after the blocking action. The variable `KILL_RUN_CMD_FIRST` controls the order—"1" to fire `KILL_RUN_CMD` before the block actions, and "0" to fire it after. You might have `KILL_RUN_CMD` send a piece of email to your pager about the block, or send an SNMP trap to a network monitor, etc. You could also have `KILL_RUN_CMD` retaliate against the remote machine in some way, but this is not recommended.

All three of the action variables can use the syntax `$TARGET$`, `$PORT$`, and `$MODE$` in the command line. These are replaced with the IP address, port, and protocol (`tcp` or `udp`) of the original probe, respectively.

Tuning Your Response

Use **BLOCK_TCP** and **BLOCK_UDP** to control response:

- "0" – do nothing except log
- "1" – trigger all defined responses
- "2" – only run **KILL_RUN_CMD**

SCAN_TRIGGER is number of probes to provoke response

One way to prevent PortSentry from taking action is to simply not set the three "action variables" discussed on the previous slide. You can also use `BLOCK_TCP` and `BLOCK_UDP` to control the PortSentry behavior to different types of scans. The default setting is "0", which means to log probes only. A setting of "2" will cause logging to happen and only `KILL_RUN_CMD` to be triggered—useful if `KILL_RUN_CMD` modifies the configuration of some other machine but you don't want to make local updates. A setting of "1" triggers all defined actions.

Normally even a single probe packet will cause PortSentry to start firing. However, you can set `SCAN_TRIGGER` to the number of packets you're willing to receive before firing. A setting of "2" will help eliminate a lot of false alarms. Note that there's really no difference between a setting of "0" and a setting of "1"—both will cause PortSentry to fire on the first packet received.

Warning Banner

- Optional **`PORT_BANNER`** is message that is displayed once PortSentry triggers
- Does not work in "stealth mode"
- Discourages attackers or enrages them?

You may optionally set `PORT_BANNER` with a message that will be displayed when an IP address that has been "blocked" by PortSentry attempts to connect to one of the ports that PortSentry is monitoring. Of course, if PortSentry has configured your local firewall to drop all traffic from the host, there's no point in doing this. On the other hand, if you're not doing active response with PortSentry ("log only" mode), this can put the person probing the system on notice that you've detected their activity. Note that `PORT_BANNER` is only displayed if PortSentry is bound to the ports in "normal" rather than "stealth" mode.

Be careful that your `PORT_BANNER` message is not overly antagonistic. Something simple like "`All unauthorized access is monitored and reported`" is sufficient. "`D00d! You sur3 are a l8m3 hax0r!`" is probably only going to provoke a more extreme response.

Starting PortSentry

- Start two daemons (one TCP, one UDP):

```
portsentry -tcp
portsentry -udp
```

- Or if you're using "stealth mode":

```
portsentry -stcp
portsentry -sudp
```

- Don't forget boot script for automatic start-up

Once you've got your configuration file all squared away, it's time to actually start the PortSentry daemon. Actually, you start two daemons—one to monitor the TCP ports you've defined and one to monitor the UDP ports. The command line invocations for the daemons are shown on the slide.

If you decide PortSentry is useful, then you'll probably want to create a boot script so that it's started automatically when the system is rebooted.

Final Thoughts

- Like AIDE, PortSentry requires some tuning work up front
- Functions best on machines that are not "well-advertised"

http://sourceforge.net/projects/sentrytools

As with any sort of IDS, PortSentry will require some tuning for your specific environment. Mostly, this involves tweaking the port lists so you don't get buried by false alarms.

The README files in the PortSentry documentation point out that running PortSentry on a very "public" machine like your external web server, name server, etc., is probably not a useful thing because these machines are getting scanned all the time. Your best bet is to have it running on an out-of-the-way machine so you can detect indiscriminate scan behavior, and on critical servers where you can very tightly define "expected" network activity.

PortSentry can be obtained from the URL shown on the slide.

506.6
Digital Forensics for Linux/Unix

SANS

THE MOST TRUSTED SOURCE FOR INFORMATION SECURITY TRAINING, CERTIFICATION, AND RESEARCH | sans.org

SEC506.6 Securing Linux/Unix

SANS

Digital Forensics for Linux/Unix

Digital Forensics for Linux/Unix

Welcome to the world of forensics. There's a tremendous amount of information that comprises this field, and we'll do our best to give you a comprehensive overview of the digital forensic specialty as we progress through this material.

Get ready, because of the tremendous amount that we have to cover, we are going to move along at a very high rate of speed.

Hal Pomeranz
hal@deer-run.com
http://www.deer-run.com/~hal/
@hal_pomeranz on Twitter

This course derived from materials originally created by John Green, used with permission.

Course Outline

- Filesystem basics
- Introduction to forensics
- What can be done to prepare?
- Importance of validating compromise
- Process for collecting evidence
- Creating images and proving integrity
- Analyzing what you've collected
- Reporting and wrap-up

Course Outline

At a very high level, these are the items that we'll be covering today.

Linux Filesystem Basics

Before we dive into forensic investigations, let's take a moment to ensure we understand the layout and structures of a filesystem.

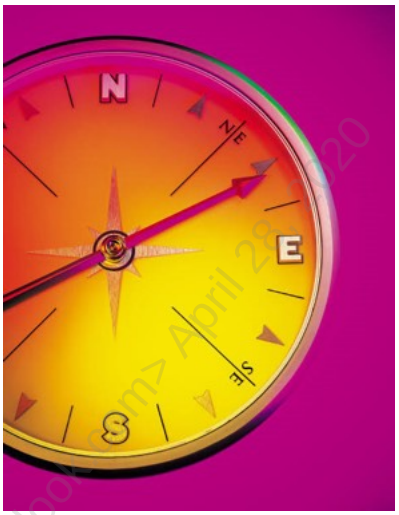
Linux Filesystem Basics

Much of your success investigating the compromise of a Linux or Unix system will depend on your understanding of the filesystem under investigation. Filesystems are quite complex, and every single one of them is different in terms of structure, features, and performance. That said, the traditional Unix filesystems (such as EXT under Linux, UFS in Solaris, FFS for the BSD OSes, etc.) have a common ancestor—the Fast File System designed by Kirk McKusick for 4.2BSD. Thus, these filesystems have certain common features that can be exploited by specialized forensic tools.

You will find less forensic tool support for other Linux filesystems, like XFS and ZFS. More information regarding XFS can be found at *https://righteousit.wordpress.com/tag/xfs/*

Filesystem Objectives

- Filesystem concepts
- Data organization
- Conceptual layers
- Critical data structures
- Sleuth Kit tools

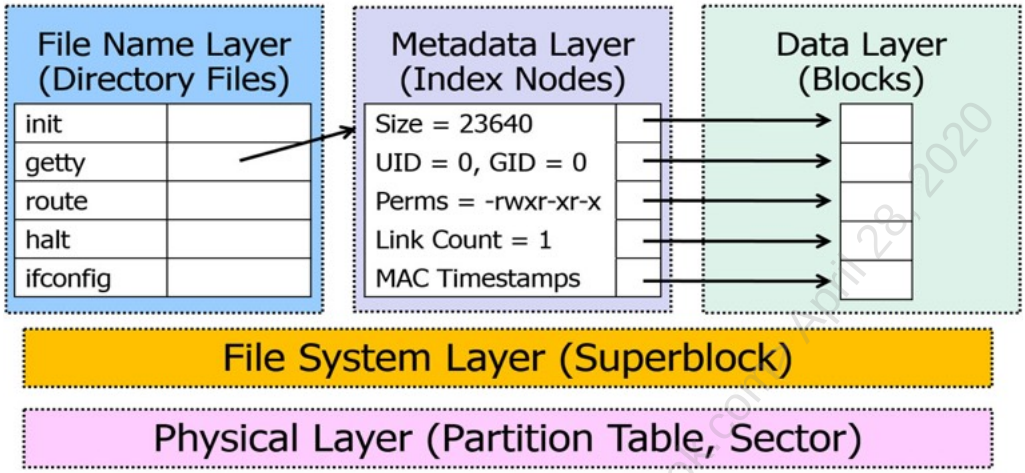


Filesystem Objectives

The filesystem is the medium in which compromises reside. In order to perform a detailed investigation of a system, it is important for the investigator to have a solid understanding of the filesystem.

This section will introduce the student to the basic concepts of Linux-based filesystems. The filesystem and its data are arranged in layers, and a few important data structures actually tie it all together. The student will be introduced to those data structures, why they're important, and some interesting tools that can be used to analyze them.

A Conceptual Model: EXT Filesystem Layers



A Conceptual Model: EXT Filesystem Layers

You're probably familiar with the "seven-layer" OSI model for describing network communications. The filesystems can be conceptualized as a "five-layer" model:

- Physical Layer: The physical drive or device and the partitions on it. Partition geometry is described by a *partition table* at the beginning of the disk—sometimes referred to as a *Volume Table of Contents* (VToC) or *disk label*. *Sectors* are the smallest unit of storage addressable by the disk controller.
- Filesystem Layer: Contains all the configuration and management data associated with the filesystems in each partition on the disk. For Unix filesystems, the primary structure of interest at this layer is an object called a *superblock*.
- The File Name Layer (AKA Human Interface Layer) is responsible for mapping human-readable filenames to metadata addresses. In Unix filesystems, this is accomplished with special *directory files* that map filenames to *index node* (*inode*) numbers in the layer below.
- Metadata Layer: Contains all of the data structures that are responsible for the definition and delineation of files. In Unix filesystems, we use objects called *index nodes* (*inodes* for short) that store metadata about files and pointers to the disk blocks that make up the contents of the file
- Data Layer: Contains the actual data units of disk storage—commonly referred to as *blocks* in Unix filesystems.

Physical Layer: Disk Partitions

- A disk can be segmented into *partitions*
  - Two types (on DOS-type disks): *Primary* and *extended*
- Each partition is treated as an independent device
- *Partition table* at beginning of the disk provides a map
- A partition usually contains a filesystem or swap

The Physical Layer: Disk Partitions

The physical layer consists of the physical disk device and the structures that define it.

A disk drive (with a filesystem) must contain at least one partition, though it may be segmented into many. The first sector of each disk contains a *partition table* or *Volume Table of Contents* (VToC, also known as a disk label). On many PC-BIOS systems, the partition information is stored in the *Master Boot Record* (*MBR*) in the first sector of the disk. The MBR only has space to store information about four partitions. These are what is referred to as the *primary* partitions on the device. When four partitions aren't enough, a primary partition can be subdivided into multiple logical or "extended" partitions. Each extended partition has a data structure at the front that looks like an MBR with slots for describing four additional partitions.

GPT (GUID Partition Tables) is an alternate disk partitioning scheme designed to overcome many of the limitations of traditional MBR-style partition tables. GPT allows for more and larger partitions and has other advanced features such as a backup partition table and error-detecting checksums.

Even though multiple partitions may exist on the same disk, the Unix operating system treats them as independent devices and performs file I/O via individual entries in the `/dev` directory.

Typically, each partition is formatted with a filesystem like EXT. Sometimes a filesystem is not formatted—for example, Unix swap partitions will typically use "raw" partitions, and some databases use raw partitions to try to improve performance.

File System Layer: Superblock

The File System Layer contains data that describes the filesystem within a partition

Unix uses a *superblock* that contains the following data:

- FS type/size, block size, number of blocks/inodes, etc.
- Modification time, last mounted on, clean/dirty status
- Pointer to inode for filesystem journal (EXT3 and above)

The File System Layer: Superblock

When a filesystem is created in a logical partition, a data structure is created at the beginning of the partition to define the attributes of the filesystem that resides there. For Unix filesystems, this data structure is called a *superblock*. The superblock contains basic filesystem information including items like the filesystem type, block size, the number of blocks and inodes in the filesystem, the number of unallocated blocks and inodes, and so on. It also contains information about the usage of the filesystem, like when it was last mounted, where it was mounted, whether it was unmounted cleanly, and so on. The superblock is always replicated, to provide fault tolerance against disk failure in the first superblock.

For EXT3 and later filesystems, the superblock contains a pointer to the inode of the filesystem journal, though this is always inode number 8. The superblock *does not* contain a pointer to the inode of the root of the filesystem, but by convention, inode 2 is reserved for the root directory (in older versions of the Unix filesystem, the "file" at inode number 1 was used to store bad block addresses).

Data Layer: Blocks

Data Layer is for storing files' contents

The basic storage unit is a *block*

- Blocks are composed of sectors (usually 8 in EXT)
- This is the smallest unit of file I/O in the filesystem

For efficiency, the blocks that make up a file are allocated consecutively when possible

The Data Layer: Blocks

The data layer is where the binary information is actually stored on disk. The smallest storage unit addressable by the disk device is a *sector* that is usually 512 bytes. However, to improve I/O performance, EXT filesystems will normally perform reads/writes in 4K chunks called *blocks*.

On modern Linux filesystems, the standard 4K block size is the minimum amount of data that will be allocated to any file. If the file is smaller than 4K, the remainder of the block is unused or "slack" space. The original BSD FFS implementations, in an effort to use disk space more effectively, subdivided the 4K block into four 1K *fragments*. Each fragment could then independently store a small file. But as disk space got cheaper, it wasn't worth the performance overhead to pack multiple small files into blocks. So, on modern Linux systems, you'll typically see the fragment size set to the same value as the block size—thus you can only have a single file per block.

When writing a large file that spans multiple blocks, the filesystem will try to allocate consecutive blocks where possible. This will increase the read efficiency because the filesystem can "read ahead" in large swaths. But this tendency also turns out to be useful when we're trying to recover deleted data. If you can locate a suspicious string in the middle of a "deleted" block of data, you may be able to recover the entire deleted file by capturing the blocks immediately before and after the "interesting" block.

Metadata Layer: Inodes

Metadata Layer stores "non-content" data about files

Uses structures are called *inodes*—every file has one

- File type
- Access rights
- Owners
- ***Timestamps***
- Size
- ***Pointers to data blocks***

The Metadata Layer: Inodes

All filesystems have structures that are used to describe or represent files. The metadata layer contains these structures. They are called different things in different filesystems, but in the Unix world, they are known as *inodes* (which is a contraction of "index nodes").

An inode contains descriptive information such as timestamps, access controls or permissions, file owner's user ID, and the file's size—everything you're used to seeing in the output of "`ls -l`" *except* for the filename. An inode also has pointers to the data blocks that make up the contents of the file.

Each inode has an address—they are simply numbered sequentially. Although inodes are typically hidden from the end-user, their addresses can be accessed ("`ls -i`") and a wealth of information can be retrieved using some specialized tools we'll be looking at shortly.

For more detailed information on the internals of inodes see:

http://computer-forensics.sans.org/blog/tags/ext4

https://digital-forensics.sans.org/blog/2008/12/24/understanding-indirect-blocks-in-unix-file-systems

Human Interface Layer: Filenames

- Partition (major/minor device number) and inode number are how the kernel tracks files
- Humans don't like accessing files via sequences of numbers
- *Directory files* associate filenames with inode numbers

The Human Interface Layer: Filenames

The Unix operating system tracks files using the inode number and the device numbers associated with the disk partition that holds the file. But human beings don't find a series of numbers convenient for naming files, so some interface layer between humans and machines is necessary.

The Human Interface (or File Name) Layer contains special filesystem objects whose purpose is to associate human-readable filenames with the inode numbers used by the OS. The "special objects" are what we call directories…

Directories

Inode contains all of the information about a file EXCEPT its name

Directories are special files that map inode numbers to filenames:

- Inode number
- Filename
- Filename length
- Size of entry

Byte Offset in Directory	Inode Number	File Name
0	84	.
16	6	..
32	1854	fdisk
48	1232	fsck
64	87	halt
80	1789	lsmod
96	1523	mkfs
112	74	mount
128	1466	reboot
144	132	umount
160	90	syslogd
176	1656	ifconfig

Directories

Directories are simply special files that associate filenames with inodes. In the traditional Unix filesystems, a directory "file" is just a sequential list of filenames along with their corresponding inode. When you list a directory, you are basically just dumping the contents of the directory "file".

Directories give the filesystem its hierarchical structure. Consider what happens in the operating system when you try to access a file like `/home/hal/.profile`:

- Remember that inode 2 is reserved for the root of the filesystem, so the filesystem driver begins by opening this "file".
- The OS reads the contents of the root directory "file" pointed to by this inode until it finds the entry for "`home`" and the associated inode with this entry.
- The OS then opens the inode from the "`home`" directory entry—this is another directory "file" and the OS scans through the contents until it finds the "`hal`" entry and its inode.
- Now we have the inode for `/home/hal`—yet another directory, so the OS has to scan through the directory to find the entry for "`.profile`".
- Finally, the OS has the inode for `/home/hal/.profile` so it can open this file and read its contents.

The Sleuth Kit (TSK)

Collection of 16 highly specialized filesystem analysis tools

- Enhances collection/analysis tools from earlier packages
- Tools are organized by filesystem layers
- Follow common syntax and naming convention

The Sleuth Kit (TSK)

The Sleuth Kit was written by Brian Carrier and has undergone a bit of an evolution. Brian originally wrote a set of tools called TCTUTILS to expand the capabilities of an older package called The Coroner's Toolkit (TCT). However, over time TCTUTILS diverged from TCT and became a standalone package. The Sleuth Kit (TSK) was born.

TSK Programs

File System Layer Tools

- **`fsstat`** Displays details about the filesystem

Data Layer Tools

- **`blkcat`** Displays the contents of a disk block
- **`blkls`** Lists contents of deleted disk blocks
- **`blkcalc`** Maps between **`dd`** images and **`blkls`** results
- **`blkstat`** Lists statistics associated with specific disk blocks

Metadata Layer Tools

- **`ils`** Displays inode details
- **`istat`** Displays information about a specific inode
- **`icat`** Displays contents of disk blocks allocated to an inode
- **`ifind`** Find which inode has allocated a block in an image

TSK Programs

Having to learn so many different tools sounds daunting, but the filesystem tools are organized by filesystem layers according to the conceptual model that we discussed. The prefix letter(s) of each tool is based on the layer name. The remainder of the name uses standard names, such as `find`, `stat`, and `ls`.

The `cat` tools display data (like the `cat` utility in Unix). The `stat` tools display statistics or information about an address, the `ls` tools list details about many addresses, and the `find` tools map between different layers.

The tools for the File System Layer start with '`fs`'. The tools for the content (data) layer start with '`blk`' (in older versions of TSK, these tools were prefixed with '`d`' but there were naming conflicts with other Unix tools like dstat so the prefix was changed). The tools for the metadata layer start with '`i`' because the original TCT tools were designed only for Unix inodes.

More TSK Programs

Human Interface (File Name) Layer

- **`fls`** Displays file and directory entries in a directory inode
- **`ffind`** Determine which file has allocated an inode in an image

Media Management (partitions)

- **`mmls`** Displays list of partitions in a disk image

Other miscellaneous tools:

Hash database tools, timeline tools, etc.

SANS SEC506 | Securing Linux/Unix 14

More TSK Programs

The Human Interface layer commands all start with the letter '`f`'. The `fls` tool allows us to list files like the `ls` command does in Unix, but has multiple additional features, including showing deleted files.

The `mmls` tool can be used to read the partition map from a bit image of an entire disk.

The remaining six tools do miscellaneous support options. The `file` tool is analogous to the Unix `file` command, determining the type of an unknown file. The `sorter` command will process files and sort them into bins of similar type. The `mactime` tool will take a specially formatted file and create a nicely formatted timeline of file modification, access, and attribute change, and finally, the hash database tools permit the comparison of the files in an image against the hashes of known good files in an attempt to reduce the number of unknown files that need to potentially be looked at.

A Layer / TSK Case Study

String search is one way of locating evidence

How do you find the file that a "hit" belongs to?

1. Use **`grep`** with a byte offset flag
2. Divide byte offset by block size to compute block number
3. Display a hexdump of block to see the string in context
4. Check if the block is allocated to a file
5. If it is, see which inode number has that block allocated
6. Search directories to find filename associated with inode number

TSK can help; let's walk through it!

A Layer / TSK Case Study

Let's take a moment to understand how the concepts that we've just presented can be used in a real-world investigation. One method investigators often use is to search a disk partition for "strings of interest." We can use `grep -b` to provide the byte offset of the string, indicating how many bytes into the partition it actually is, but as you can imagine, that doesn't do us a lot of good.

If I find a string that might indicate evidence, then I want to know if it resides in a file and if so, what is the file's name? It is possible to figure this out, but it takes a bit of work and a progression through the different layers that we've seen.

Let's do an example, and introduce some tools along the way.

Partitions and Mount Points: Viewing Physical Layer Data

```
# fdisk -l
Disk /dev/sda: 20.0 GB, 20003880960 bytes
255 heads, 63 sectors/track, 2432 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot      Start         End      Blocks   Id  System
/dev/sda1   *           1          13      104391   83  Linux
/dev/sda2              14        2367    18908505   83  Linux
/dev/sda3            2368        2432      522112+  82  swap

# mount
/dev/sda2 on / type ext3 (rw)
none on /proc type proc (rw)
none on /sys type sysfs (rw)
none on /dev/pts type devpts (rw,gid=5,mode=620)
/dev/sda1 on /boot type ext3 (rw)
none on /dev/shm type tmpfs (rw)
```


Partitions and Mount Points: Viewing Physical Layer Data

For any disk or image that you might be investigating, information from the physical layer is very important. Many of the tools we will use will need to know what the filesystem is, and where a particular partition may be mounted.

In this case, we have used `fdisk -l` to safely display the disk's partition table. We also used the `mount` command to see where these partitions were actually mounted into the filesystem. Although the `df` command could be used to list partition information, the advantage that the `mount` command provides is that, in addition to partition and mount information, it also shows us the filesystem type as it is mounted on the system.

Of course, in a normal forensic investigation, we wouldn't be running these sorts of commands on the live system. Instead, we would use the `mmls` tool from TSK to access the partition information from a *disk image* that was created from the original drive. We'll discuss disk imaging and meet `mmls` later in this course.

Also I am glossing over some complexity because modern Linux systems use Logical Volume Management (LVM) and often full disk encryption. For more details see:

https://digital-forensics.sans.org/blog/2010/10/06/images-dmcrypt-lvm2

Viewing Filesystem and Block Size

```
# fsstat /dev/sda2
FILE SYSTEM INFORMATION
--------------------------------------------
File System Type: Ext3
Volume Name: /
Last Mount: Sat Aug  6 21:01:32 2004
Last Write: Sat Aug  6 21:01:32 2004
Last Check: Mon Jun 14 09:07:26 2004
Unmounted properly
:       :           :
CONTENT-DATA INFORMATION
--------------------------------------------
Fragment Range: 0 - 4727125
Block Size: 4096
Fragment Size: 4096
```


Viewing Filesystem and Block Size

Once we have the physical layer data, we'll need to get a feel for the File System Layer information. Although all of the data here is interesting to us, the most important for the purpose of this walkthrough is the File System Type (Ext3), and the Block Size (4096).

```
# fsstat /dev/sda2
FILE SYSTEM INFORMATION
--------------------------------------------
File System Type: Ext3
Volume Name: /
Last Mount: Sat Aug  6 21:01:32 2004
Last Write: Sat Aug  6 21:01:32 2004
Last Check: Mon Jun 14 09:07:26 2004
Unmounted properly
Last mounted on:
Operating System: Linux
Dynamic Structure
Compat Features: Journal, Ext Attributes, Dir Index
InCompat Features: Filetype, Recover,
Read Only Compat Features: Sparse Super,

CONTENT-DATA INFORMATION
--------------------------------------------
Fragment Range: 0 - 4727125
Block Size: 4096
Fragment Size: 4096
```

Where's Waldo? Block 170338!

```
# grep -abi waldo /dev/sda2 > /tmp/found_waldo.txt
# less /tmp/found_waldo.txt
:   :   :
689702819:^@^@<8D^@submit@bugs.kde.org^@(c) 2003 Waldo
Bastian^@Author^@bastian@kde.org^@No
696076095:Do you want to save the changes or discard
them?^@editor^@0.5^@submit@bugs.kde.org^@KDE Menu
Editor^@kmenuedit^@bastian@kde.org^@Waldo
Bastian^@sandrini@kde.org(C) 2000-2003, Waldo 697911472:Waldo
697911478:Waldorf
   :    :    :
```

697911472(byte offset) / 4096(blocksize) = 170388(block)

SANS SEC506 | Securing Linux/Unix 18

Now, Where's Waldo? Block 170338!

At this point, we've got all of the information we need to start the fun part. In this example, we are going to use the string "Waldo" because it's more fun, but in an investigation, you will be searching for strings that are commonly associated with malware, stolen or illicit data, etc.

To begin, we can use the `grep` command with some specific flags: The `-a` flag says to treat everything as ASCII data (so `grep` doesn't choke on the raw binary filesystem data), the `-b` flag says to provide a byte offset at the beginning of each matching line, and the `-i` flag says to ignore case when searching for the string in question (you've probably used this option many times). To make things easy for this example, I have redirected the output to a temporary file.

When that finishes, we can examine the resulting file to see if Waldo was found, and indeed, he was—many times. Choosing an interesting line, I record the byte offset.

Now to calculate which disk block that occurrence of Waldo was in, I divide the byte offset (697911472) by the block size (4096) we got from the `fsstat` output on the previous slide and truncate the result to an integer (if you use the `expr` command in Unix, it will automatically give you the integer output). My result is that Waldo is hiding in block number 170388 on the disk.

Viewing Data Layer Data

```
# blkcat -h /dev/sda2 170388
 :    :    :
2144  0a77616b 656e0a77 616b656e 65640a77   .wak en.w aken ed.w
2160  616b656e 696e670a 77616b65 730a7761   aken ing. wake s.wa
2176  6b657570 0a77616b 696e670a 57616c62   keup .wak ing. Walb
2192  72696467 650a5761 6c636f74 740a5761   ridg e.Wa lcot t.Wa
2208  6c64656e 0a57616c 64656e73 69616e0a   lden .Wal dens ian.
2224  57616c64 6f0a5761 6c646f72 660a5761   Wald o.Wa ldor f.Wa
2240  6c64726f 6e0a7761 6c65730a 57616c66   ldro n.wa les. Walf
2256  6f72640a 57616c67 7265656e 0a77616c   ord. Walg reen .wal
2272  6b0a7761 6c6b6564 0a77616c 6b65720a   k.wa lked .wal ker.
2288  77616c6b 6572730a 77616c6b 696e670a   walk ers. walk ing.
2304  77616c6b 730a7761 6c6c0a57 616c6c61   walk s.wa ll.W alla
:    :    :
```


Viewing Data Layer Data

Before I go too far, I want to know if this is an "interesting" Waldo. It is often very helpful to see the string of interest in the context of surrounding data, and then make a decision about whether it may be related to the case under investigation.

So, we can use information from the data layer and a neat tool called `blkcat`. `blkcat` will display the contents of a disk block to STDOUT. Be careful though—listing binary contents to STDOUT can get messy. Instead, we'll make `blkcat` give use results in a hexdump-like display format using the `-h` flag. You must also specify the partition name and block number on the `blkcat` command line.

In the slide above, you see the hexdump results from `blkcat`, and in fact, Waldo appears twice! It looks like in this case, Waldo may just be part of a dictionary or wordlist, but we'll keep going with our example anyhow.

Allocated to a File? Which One? Viewing the Metadata Layer

Is it allocated to a file?

```
# blkstat /dev/sda2 170388
Fragment: 170388
Allocated
Group: 5
```

Which inode does it belong to?

```
# ifind -d 170388 /dev/sda2
69739
```


Allocated to a File? Which One? Viewing the Metadata Layer

Just because we found a block that Waldo resides in, doesn't mean that this block is still allocated to an existing file. It could've been deleted long ago. So, to check to see whether or not it belongs to an existing file, we can query the metadata layer to see if there are any inodes that have this disk block allocated to them.

The `blkstat` command provides allocation statistics about a given block on disk. It requires the same parameters that we saw earlier: A filesystem image or partition, and the address of a disk block. Here, we see that it is indeed allocated to an existing file.

Remember, the goal of this exercise was to find the name of the file that contains this instance of the string Waldo, so we're going to need to figure out which inode has this block allocated. For this, we'll use the `ifind` command. Given a block number, this command will search the inodes to find which one has that block allocated. Again, the parameters are very similar to earlier commands we've seen. The exception is that to specify the disk blocks address, we must use the `-d` flag.

`ifind` determines that the block is allocated to inode number 69739.

How About the File Metadata? Viewing Metadata Layer Data

```
# istat /dev/sda2 69739
inode: 69739
Allocated
Group: 4
uid / gid: 0 / 0
mode: -rw-r--r--
size: 409305
num of links: 1

Inode Times:
Accessed:       Tue Feb 17 19:47:53 2004
File Modified:  Tue Feb 17 19:47:53 2004
Inode Modified: Sun Jun 13 23:13:18 2004

Direct Blocks:
170290 170291 170292 170293 170294 170295 170296 170297
:      :      :
```


How About the File Metadata? Viewing Metadata Layer Data

Now that we have the inode number that allocated the block OUR Waldo resides in, it might be informative to take a look at the data contained in an inode to get a feel for the size of the file, the owner, permissions, and timestamps. To do this, we can use the `istat` command. We can see that it is quite a large file, owned by root, and readable by everyone. It has been quite a while since the file was accessed or modified though.

TSK also provides a tool called `icat` that would enable us to dump the contents of this file. However, we're more interested in figuring out what file this actually is, so let's proceed.

Viewing Human Interface Layer Data. And Now the Filename …

```
# ffind -a /dev/sda2 69739
/usr/share/dict/linux.words
```


Viewing Human Interface Layer Data. And Now the Filename …

Finally! We've reached the point that we can determine the name of the file where Waldo was hiding. Now that we have the inode number, we can use the `ffind` command to determine the filename. Given an inode number, `ffind` looks through the directory entries, finds the inode number, and displays the corresponding filename.

The `-a` flag tells `ffind` to display all of the entries or names associated with this inode—after all, there may be multiple hard links pointing to a single inode. However, if you look carefully at the `istat` output on the previous slide, you'll see that the "num of links" value is 1, so as soon as we find the first directory entry that points to this inode, that's it. But the `-a` option will force `ffind` to scan the entire filesystem rather than stopping on the first hit—a waste of time in this case.

So, Waldo was hiding in the file: `/usr/share/dict/linux.words`. Whew!

Exercise 1: Looking for Love

- Purpose: Understanding the conceptual layers
- Search root filesystem for string " love " (note the spaces)
- Pick a "hit" and work back to the filename
- Use the tools that we just covered

Exercise 1: Looking for Love

Exercises are in HTML files in two places: On the course USB media, and in the home directory of the SANS user in the lab virtual machine. These two locations contain identical copies of the exercise files—it just depends on whether you prefer to look at them from a web browser inside the lab VM (because you're working in full-screen mode) or from your host operating system.

1. Open your web browser
2. Select "File … Open …" from the browser menu (Ctrl-O is the usual keyboard shortcut)
3. This is Day 6, Exercise 1, so navigate to *... /Exercises/Day_6/index.html* and open this HTML file
4. Click on the hyperlink, which is the title of Exercise 1
5. Follow the instructions in the exercise

Introduction to Forensics

Introduction to Forensics

Forensic investigations are not rocket science. However, they do require a firm understanding of the operating system and filesystem of the victimized computer. It also helps a tremendous amount to have a grasp of the techniques that attackers use to compromise systems and conceal their activities. Finally, knowing what tools are available to help the process along is invaluable.

Forensics requires patience, tenacity, and a little bit of luck. Combine those qualities with a bit of process and a standardized methodology and you're well on your way to being a forensic analyst.

Although we'll be walking through the forensic process in a day, be aware that the investigation of a compromised system may easily take 40 or even 80 hours of work. Now that's patience and tenacity!

Forensics in a Nutshell

- Validate compromise
- Evidence seizure
- Investigation and analysis
- Reporting Results

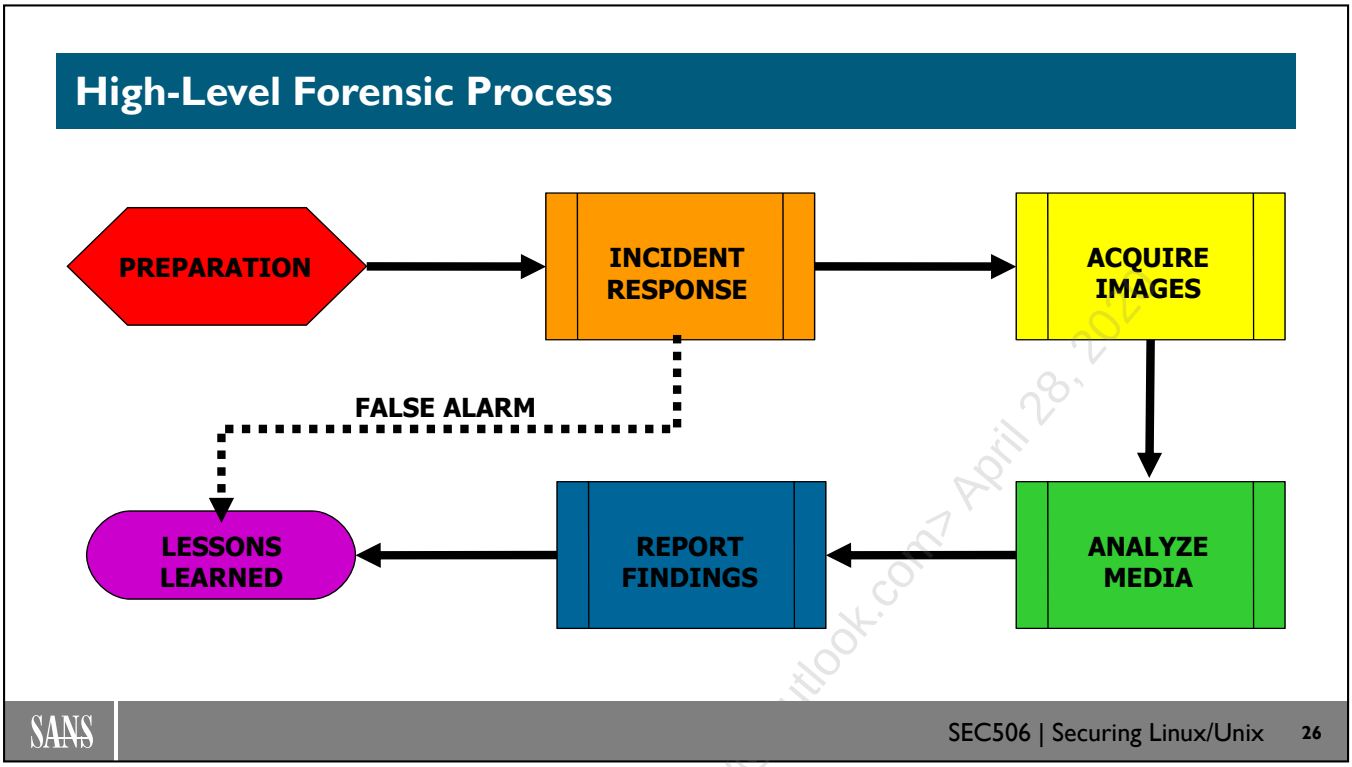
Forensics in a Nutshell

Computer forensics is more than just analyzing blocks of data. It is the effective gathering, examination, and reporting of your actions and findings. A seasoned investigator knows that if one step is overlooked, his case will not yield the results he or she may desire.

Evidence seizure generally occurs during the incident response phase where you must verify the incident, but you also begin your work to collect volatile and non-volatile data. Data that is volatile is lost if the system is shut down or rebooted—for example, a memory dump of a process that contains key IP addresses of the attacker or subject. Non-volatile data can be collected after the system is turned off—hard drive images, backups, etc. There may also be information on other systems—IDS and firewall logs, for example.

Investigation and analysis occur when the investigator takes what is collected and analyzes it to form a clear picture of the incident. This analysis uses tools and techniques that require data recovery, piecing together the puzzle of what happened, and forming a timeline of events.

Reporting your results becomes the most important step. Without accurate reporting, the investigator often finds himself unable to find anyone willing to prosecute his case or take action. Without action, why perform the investigation? Reporting is key.



High-Level Forensic Process

Here, you see a simplified flowchart of the forensic investigation process. We'll discuss each of these steps in detail throughout the course of the material, but the process is worth mentioning here at a high level. Being adequately prepared to respond is absolutely critical. If you spend enough time in this proactive phase, you can greatly reduce the stress that you encounter and the chance of making costly mistakes during the reactive process of incident response.

It is important to mention that not every incident is an intrusion or compromise, so we have incorporated a dotted line to indicate that false alarms do occur. In these cases, the incident response team is activated to handle what appears to be an incident, but it later turns out to be something less serious—such as an improperly configured system or over-zealous user/administrator.

Therefore, the first step of incident response is usually to validate that an intrusion has occurred. Beyond that, incident response is a carefully choreographed dance that balances the investigation of a system with the requirement of minimized data loss due to the investigation. Image acquisition is the process of creating true bit images of the disks and removable media that may be associated with the system in question. It is an art form unto itself, simply because of all of the different types of storage a responder might encounter in the field. Once these images have been acquired, the original evidence (system disk) can be locked in a container for safe keeping.

You will conduct your investigation against these images, performing media analysis by examining the images for small pieces of evidence that will ultimately help you to reconstruct what happened in the past on a system. The two final pieces are the ones that are most often overlooked. The creation of an accurate and effective incident report, and incorporating any lessons learned back into the security processes and policies of the organization.

Major Challenges

- Rapid action
- Recording the scene without disturbing it
- Maintaining evidence integrity
- Compromised software tools and/or kernel

Major Challenges

As I see it, there are several major challenges facing the forensic investigator.

The first challenge is rapid action. When an incident occurs, an immense amount of pressure is placed on the incident handling team to investigate and recover from the compromise as soon as possible. This can make it very difficult, if not impossible, to collect evidence in a thorough manner.

Another problem is making sure that once you've collected and analyzed the data, it can be submitted as evidence in court and will hold up under scrutiny. Remember, the defense's goal is to convince just one juror that you didn't have all of your I's dotted and T's crossed when you were investigating the incident.

A key piece of the investigation strategy is to avoid disturbing the crime scene. The question is, "How does one record the scene of a computer incident without disturbing it?" Merely by logging on to the system, the "state" of the computer is altered.

And finally, how do you investigate a computer, using the tools that may themselves be compromised, or changed to hide the intruder's activities and presence. I hope to show you how to overcome these obstacles during the course of the lecture today.

Guiding Principles

- Work to minimize evidence loss
- Take great notes in excruciating detail
- Collect all the evidence that you can
- Analyze everything you collect
- Prove that evidence/process integrity has been maintained
- Learn lessons from each incident

Guiding Principles

There are a few basic principles to ensure success when performing a forensic analysis on a computer. The number one rule—attempt to minimize the loss of evidence. There is no way to NOT affect the system. You will lose evidence. The key to managing this problem is understanding what effects your actions have and what data is preferable to lose. Armed with this knowledge, you can take steps in order to minimize that loss and maximize the data available for collection.

Record everything. Record what you do. Record computer memory, disks, and everything else about the computer. Record the scene. There is nothing that you should not record while at the scene. Remember, once you leave the scene or power down the system, much of the evidence is gone.

Analyze all the data that you collect. You'll likely end up with a huge amount of data, but don't let that deter you. Forensics can be a tedious process; thorough investigation of a single system can take days or even weeks depending on what you find when you start looking.

Maintain the integrity of the evidence. This is a critical step. You will learn what you can do to prove that the integrity of the evidence has not been lost, and at what points in the investigation you can apply these concepts.

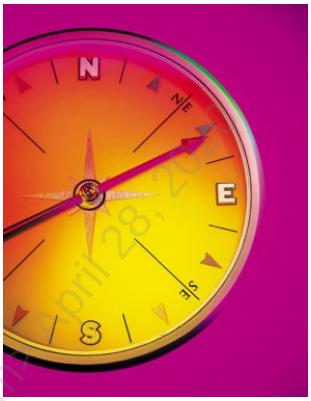
Forensic Preparation

Forensic Preparation

Much of the work involved in incident handling and forensic investigations can, and should be done ahead of time. These proactive measures will help you every step of the way when actually responding and investigating an incident. Having plans, call lists, response policies, an adequate detection infrastructure, and a trusted set of tools may ultimately make the difference between a successful response/investigation and one that is botched.

Preparation Objectives

- Methodologies guide you
- Infrastructure alerts you
- Forensics labs enable you
- Toolkits assist you
- Policies protect you



Preparation is the key to effective and efficient DFIR

Preparation Objectives

In this section, we'll take a quick look at some of the things you should consider doing ahead of time, and during the non-response hours of your day. Understand that each of the areas on the slide above provide great benefits during the investigation.

If the case you are investigating goes to court, one of the things you will be questioned about is the investigation methodology that was used. Obviously, a well-thought-out and planned methodology is superior to an ad-hoc response. A good methodology will be a general and flexible framework that provides guidance on the steps of the investigation. It must be flexible though, to account for the fact that no two incidents are the same, and so, responses may need to vary slightly. Be careful, though, if your organization has developed and documented a rigid methodology, then you had better follow it, or else that could become a point of contention.

If correctly configured, your network and computing infrastructure contribute supporting data to the investigation. This primarily revolves around network time, audit logs and IDS/firewall captures, but can also include the creation of a trusted baseline image or the creation and storage of a hash database.

The forensic lab is your office though it need not have walls. It enables you to perform your job. For the purposes of this course, our forensic lab will be limited to the scope of a bag of critical items that are typically needed for response, including a laptop system that has been loaded with the forensic tools of your choice.

Much of the forensic process could be extremely laborious, repetitive, and prone to error if left to the very low-level tools that exist to manipulate the data on a disk. Luckily, some really smart people have created some great tools over the last few years that all but eliminate these hardships.

Finally, in a world where critical decisions that may adversely affect business must be made quickly, response policies are invaluable for protecting you: The responder/investigator.

Two Key Questions

1. Will all incidents lead to prosecution?
2. Who's on the incident response team?

Two Key Questions

Before you start your planning, it's worth considering the answers to two key questions. First, as a matter of general policy, does your organization go into incidents assuming that the end result will be prosecution via the legal system? If yes, then you will tend to be much more careful about collecting, storing, and examining evidence than if your goal was simply to understand the compromise and get your systems functioning again as quickly as possible. Prosecution also means additional time away from work for the investigators, system administrators, and other potential witnesses and may involve additional costs for storing and producing evidence.

The other question to answer is, "Who is involved in handling incidents in your organization?" You need to know this information so that (a) you can publish it within the organization so people know who to call, and (b) you can as a matter of policy grant the individuals involved in incident response extraordinary powers when dealing with incidents. Answering this question also lets you document who's *not* involved in handling incidents, which can help stop well-intentioned but unauthorized/untrained personnel from messing up your investigations.

Do You Have a Plan?

- Do you have an incident response plan in place?
- Plans should be different for different types of incidents
  - External Incident
    Intrusions, viruses, denial-of-service, theft of service
  - Internal Incidents
    Intellectual property theft, malicious intent, policy abuse
- Documented as part of your incident response policy

Do You Have a Plan?

Does your organization have a specific computer incident response management and policy guide? While it may be boring for the typical person to work out the details of these plans, nothing is more important than this document that will outline who does what, and which actions can and will be taken during the course of an incident.

Incidents create chaos; you need to mitigate that chaos by having a well-thought-out plan prior to the incident. There should be different plans for different categories of incidents: External intrusions, insider theft, virus/worm outbreaks, DDoS attacks, etc. Work with the person responsible for your corporate disaster recovery and business continuity planning; they will be grateful for the help and may have a lot of valuable advice. Use press reports about incidents at other organizations to guide table-top exercises for your own incident response teams. This can be one good way to develop plans for different types of incidents.

While external incidents get the majority of the press, and will often consume much of an incident response team's time, it's the internal incident that usually does the most damage to an organization. The key thing to understand is that your response will likely be drastically different in these two cases. The external incident is likely to elicit the "no holds barred" style response: Get in there, get it cleaned up, and get us back into business as quickly as possible. On the other hand, the responding to, and investigating an internal incident is likely to involve an entirely different set of players (Human Resources for example) and will be conducted in much more secrecy perhaps over a longer period of time.

Policies Remove Doubt

Responders often called-in outside of normal working hours

- Decision Makers may be hard to contact
- IR Team may need elevated authority during incident
- Clearly document this authority

Incidents are high-stress situations

- *Make incident handling decisions ahead of time and document them to avoid paralysis*

Policies Remove Doubt

Unfortunately, attackers don't compromise systems on a 9 a.m. to 5 p.m. basis. In fact, it seems as if the most major incidents occur around 5 p.m. on a Friday. It's a common running joke: "It's 5 p.m., do you know where your compromise is?"

During these non-business hours, it can be difficult to contact those people in the organization who are normally responsible for making critical business decisions. Like what? Like taking your e-commerce web server offline, and essentially disabling a major income vector of the business.

Forward-thinking organizations plan for this eventuality and provide a lead responder with elevated levels of authority when the normal call escalation process is exhausted. If this type of countermeasure hasn't been put into place and documented, it can be a frightening experience for the lead responder: Do I pull the plug and bring the business to a halt? Do I wait till I can get authorization and risk the loss of reputation or intellectual property? Will I lose my job as a result of my actions? You can understand the stress that such a situation creates. That stress usually results in responder paralysis.

However, if an escalation process has been documented and includes the ability of the lead incident handler to make those tough calls when a decision maker can't be reached, then one can feel a bit more confident in proceeding. You see, this works even better if incident response plans have been put into place, and you have a general idea of how the organization SHOULD respond given an incident of type X.

If management is uncomfortable giving that kind of authority to a single person, then perhaps a majority two out of three votes of the three senior members of the incident response team would be an acceptable alternative.

Prepare Your Infrastructure

Sync system time across the enterprise
- Use Network Time Protocol
- Consider GMT if you are a global organization

Ensure logs are being generated and reviewed
- Firewalls, IDS, email, file servers, system logs

Make sure backups are created and safely stored
- Critical servers and tertiary servers

Create hash databases for deployed platforms

Prepare Your Infrastructure

Before anything ever happens on your network, you can work to ensure that your infrastructure is prepared to perform its role in the investigation process.

Make sure every machine on the network is time-synched. If your organization is geographically dispersed or even global, then it is recommended that your system times are reflected in GMT (Greenwich Mean Time) or synched to the location of corporate headquarters.

You should have a policy in place to regularly store and backup your network log files. This is especially critical in the case of your firewalls and IDS (Intrusion Detection System) log files. A properly-tuned IDS and firewall log will generally be easy to store for a long time since the log files will be smaller.

Ensuring your network servers have a backup is very important. If you have to take a server offline for an investigation, what will take over its function in your infrastructure? Having a backup is a good idea and usually, most larger companies do as it is truly the LAST line of defense and the last resort of response.

If I have a critical system and create a hash for each binary on that system, I can use those hashes to detect changes to any of those system binaries—indicating a possible compromise. This is exactly how file integrity tools work (e.g., Tripwire, AIDE, etc.), and is an excellent idea to implement organizationally.

Pre-Built Forensic Distros

SIFT Workstation
http://digital-forensics.sans.org/community/downloads

Paladin
http://sumuri.com/product-category/paladin/

CAINE
http://www.caine-live.net/

DEFT
http://www.deftlinux.net/

Pre-Built Forensic Distros

You'll also want a collection of specialized tools to help you investigate the incident. There are several different distros available that are specifically designed for incident response and forensics. SIFT is a Linux distro originally created by Rob Lee for the SANS Forensics Curriculum, but now a fully functional forensic distribution based on Ubuntu Linux. CAINE, DEFT, and Paladin Linux are other well-known Linux Forensic distros.

Kali Linux (formerly BackTrack) is generally thought of as a hacking or penetration testing platform, but it also happens to contain many forensic tools as well.

Incident Response Process

Incident Response Process

The practice of incident response is an art form unto itself, regardless of the incident. In theory, one can certainly extract forensics from the incident response process and look at it alone, but that provides little help or guidance to those who must implement the knowledge that they gain.

Instead, for the purposes of this course, we have chosen to look at how incident response and forensics are used in tandem, in the real world. The incident response team is likely the same team that will collect and ultimately analyze the evidence, so we will be approaching the problem holistically. Though there is no true standard incident response process when it comes to the forensic side of things, there are certainly some acknowledged best practices and guidelines. This course is an attempt to present these in a Unix context, at a high level.

In general, the incident response process as it applies to forensics involves the collection of evidence. This collection must be performed in a forensically sound manner. Care must be taken to collect evidence that accurately reflects the state of the system as it was discovered. This is a time-critical process, and one in which the order of collection is of utmost importance.

Although there are specialized tools for the collection and analysis of digital evidence, many of them are operating system commands that Unix system administrators should already be familiar with.

"Houston, We *Might* Have a Problem!"

```
Red Hat Linux release 7.2 (Enigma)
Kernel 2.4.7-10 on an i686

This server is operated for authorized users only. All use
is subject to monitoring. Unauthorized users are subject to
prosecution. If you're not authorized, LOG OFF NOW!

localhost login: root
Password:
Last login: Wed Aug  6 11:16:40 on tty2
[root@localhost root]# (swapd) uses obsolete (PF_INET,SOCK_PACKET)
eth0: Promiscuous mode enabled.
device eth0 entered promiscuous mode
NET4: Linux IPX 0.47 for NET4.0
IPX Portions Copyright (c) 1995 Caldera, Inc.
IPX Portions Copyright (c) 2000, 2001 Conectiva, Inc.
NET4: AppleTalk 0.18a for Linux NET1.8
eth0: Promiscuous mode enabled.
eth0: Promiscuous mode enabled.
_
```

Ewww, is this due to an attacker OR an administrator doing what s/he shouldn't?

"Houston, We *Might* Have a Problem!"

Imagine being called to the console of this system. "Hey, John! Come take a look at this; something weird is on the screen!" The first thing noticed is that the screen is displaying a message indicating the "*device eth0 entered promiscuous mode*".

When a Network Interface Card (NIC) enters promiscuous mode, traffic on the local LAN segment is being sniffed. The knee-jerk response is that this is bad, really bad. But, has the system actually been compromised? It could be one of your network admins running `tcpdump`. Or is it an overzealous administrator or user playing with new toys? Either way, it's an incident, but a sniffer as part of a compromise is much worse.

A quick interview with the users and administrator would probably be enough to say for certain whether or not this system has been compromised. However, this is a good time to introduce the fact that all incidents and suspected compromises should be validated before activating the entire incident response team.

Incident Response Methodology

1. Open "case file", snapshot scene
2. Validate compromise
3. Collect different types of evidence
4. Analyze memory
5. Review filesystem timeline
6. Recover filesystem artifacts, deleted data
7. Create incident report
8. Document and apply lessons learned

Incident Response Methodology

The overall forensic investigation methodology will remain the same from operating system to operating system. You will probably conduct some, if not all, of these steps in every investigation. Despite the tool that you use, your overall process will and should remain the same. If you use a commercial tool versus and open-source toolset, you will still gather evidence, obtain investigation leads, and perform data recovery. Regardless of what your tools are, your methodology will aim to reconstruct what has occurred in the past on a system, and hopefully, point to the intruder.

Step 1: Create "Case File"

Before touching the system, document what you know:

- Interview users and administrators
- Suspicious network or system activity?
- What was the tip-off?

What about the victim system?

- Where did the system come from?
- What is its purpose?
- How is it configured? (OS, apps, network)
- Photos, state of computer, what is on the screen?

Step 1: Create "Case File"

This is pretty straightforward. Just like in a museum: "DON'T TOUCH!" and especially don't let others touch.

You aren't looking for actual fingerprints in the local area. You are trying to limit things that could "change the state" of the system. Even choosing to limit activity on the system ALSO changes the state of the system. So, you are trying to keep the changes on the system to a minimum. These methods might not be the best, but it is ONE way you could go about completing this goal!

In general, describe the system you are analyzing. Where did you acquire the system? What is/was it used for? What is the configuration of the system (OS, network)? Include any other information you feel may be necessary to perform the investigation.

Your system description will affect the way your investigation is executed. If this machine is a critical server, you may not be able to shut the server down. If the machine is a workstation, you may need to determine what the workstation is utilized for. Being able to predict the type of information that the system stores would aid in your ability to perform your collection and your analysis.

Step 2: Validate the Compromise

GOAL: Find evidence system has been compromised *before* activating the entire incident response team

Validation sometimes easy:

- Defaced web pages
- Obvious DoS attacks
- IDS captures

SANS SEC506 | Securing Linux/Unix 40

Step 2: Validate the Compromise

Occasionally, validation is a no-brainer. A pink pony on the homepage of your businesses website is a big clue that something has gone very wrong. IDS captures may provide conclusive evidence that a system's security has been breached, or that a DoS attack is the reason your site has lost internet connectivity. However, the clues that something has gone wrong are not always this obvious. In fact, they rarely are.

Therefore, once you have been notified of a suspected incident and have opened a case file, it is wise to work to validate that a system has actually been compromised. There will be numerous incidents where the compromise wasn't really a compromise after all—and there's the incident response team, all psyched up and nowhere to go.

What Are We Looking For?

Example evidence might include:

- Backdoors on open ports
- Hidden files or directories (rootkit)
- Unusual processes
- Promiscuous mode NIC
- Altered files (password, shadow, binaries)
- Suspicious (or non-existent) log entries

What Are We Looking For?

Basically, the idea at this stage is to take a quick look at the system—as non-invasively as possible—and try to find signs that something has gone wrong. Some good ideas of places to look are listed on this slide, and we'll talk about a few of them in more detail in the upcoming slides.

ps Can Be a Good Start

```
USER    ...     STAT START   TIME COMMAND
root    ...     S    Apr15   0:04 init
root    ...     SW   Apr15   0:00 [kflushd]
root    ...     S    Apr15   0:00 gpm -t ps/2
xfs     ...     S    Apr15   0:00 xfs -droppriv -daemon ...
root    ...     S    Apr23   0:00 syslogd -m 0
root    ...     S    Apr23   0:00 klogd
root    ...     S    Apr23   0:00 crond
root    ...     S    Apr23   0:00 inetd
root    ...     S    Apr23   0:00 (nfsiod)
  :               :          :          :          :
root    ...     S    Apr24   0:00 /sbin/mingetty tty6
root    ...     S    Apr24   0:00 /usr/bin/kdm -nodaemon
root    ...     S    Apr24   0:01 /etc/X11/X -auth /usr/...
root    ...     S    12:33   0:00 -sh
root    ...     R    12:41   0:00 ps -auxww
```


ps Can Be a Good Start

A careful examination of the output from an appropriately formed `ps` command can show quite a few things. Obviously, this machine was booted on April 15 (look at the time on the `init` process), but many of the logging daemons were apparently restarted on the 23rd. Why? Maybe it was a system admin doing legitimate maintenance, but maybe it was an attacker making configuration changes.

Do you recognize all of the processes? What about that suspicious "`(nfsiod)`" process? This machine is not an NFS server, so that process shouldn't exist. Knowing the processes that should be running on the OS versions in your organization can be useful.

Backdoor Evidence

```
COMMAND     PID  USER   FD   TYPE DEVICE SIZE NODE NAME
smbd       3137     0   6u   IPv4   4571       TCP *:2003 (LISTEN)
smbd       3137     0  16u   IPv4    976       TCP *:443 (LISTEN)
smbd       3137     0  17u   IPv4    977       TCP *:80 (LISTEN)
(swapd)    3153     0  16u   IPv4    976       TCP *:443 (LISTEN)
(swapd)    3153     0  17u   IPv4    977       TCP *:80 (LISTEN)
initd     15119     0   3u   IPv4  15617       TCP *:65336 (LISTEN)
initd     15119     0   5u   IPv4  15619       TCP *:65436 (LISTEN)
initd     15119     0   6u   IPv4  16157  TCP 192.168.1.79:65336->213.154.118.200:1188
initd     15119     0   9u   IPv4  15909  TCP 192.168.1.79:1146->199.184.165.133:6667
initd     15119     0  12u   IPv4  16191  TCP 192.168.1.79:1149->64.62.96.42:6667
xopen     25239     0   8u   IPv4   9972       UDP *:3049
xopen     25239     0  17u   IPv4    977       TCP *:80 (LISTEN)
xopen     25241     0   8u   IPv4  12302       TCP *:3128 (LISTEN)
xopen     25241     0  16u   IPv4    976       TCP *:443 (LISTEN)
lsn       25247     0  16u   IPv4    976       TCP *:443 (LISTEN)
lsn       25247     0  17u   IPv4    977       TCP *:80 (LISTEN)
```


Backdoor Evidence

This `lsof` output shows evidence of multiple backdoors. The Samba daemon `smbd` should not be listening on those ports. And `lsn`, `xopen`, and `swapd` are somehow using ports 80 and 443 as well. This only happens if these are related processes—possibly `fork()`ed copies that have changed their process name.

`initd` has connections open to port 6667, which is usually used for IRC. This wouldn't be normal for our system, either.

Unusual Processes

NAME	PID	PORTS	WORKING DIRECTORY
smbd	3137	TCP 80, 443, 2003	/tmp/sand
(swapd)	3153	TCP 80, 443	/usr/bin
initd	15119	TCP 65336, 65436	/etc/opt/psybnc
xopen	25239	TCP 80, 443, UDP 3049	/lib/.x/s
xopen	25241	TCP ports 80, 443, 3128	/lib/.x/s
lsn	25247	TCP ports 80, 443	/lib/.x/s

Examination of **ps** and **lsof** output yields results summarized above...

Unusual Processes

```
140 S root       845     1  0  69   0    -   814 do_sel Aug09 ?          0:00 smbd -D
PWD=/ HOSTNAME=localhost.localdomain CONSOLE=/dev/console PREVLEVEL=N CONFIRM= runlevel=3
MACHTYPE=i386-redhat-linux-gnu LANG=en_US SHLVL=2 previous=N SHELL=/bin/bash HOSTTYPE=i386
OSTYPE=linux-gnu HOME=/ TERM=linux PATH=/sbin:/usr/sbin:/bin:/usr/bin:/usr/X11R6/bin
RUNLEVEL=3 INIT_VERSION=sysvinit-2.78 _=/sbin/initlog

040 S root      3137     1  0  69   0    -   475 do_sel 13:33 ?          0:03 smbd -D
PWD=/tmp/sand HOSTNAME=localhost.localdomain MACHTYPE=i386-redhat-linux-gnu SHLVL=5
SHELL=/bin/false HOSTTYPE=i386 OSTYPE=linux-gnu HOME=/ TERM=dumb
PATH=/usr/local/bin:/bin:/usr/bin _=/usr/bin/smbd -D

100 S root      3153     1  0  69   0    -   416 wait_f 13:33 ?          0:00 (swapd)
PWD=/usr/bin HOSTNAME=localhost.localdomain MACHTYPE=i386-redhat-linux-gnu
OLDPWD=/tmp/sand SHLVL=5 SHELL=/bin/false HOSTTYPE=i386 OSTYPE=linux-gnu HOME=/ TERM=dumb
PATH=/usr/local/bin:/bin:/usr/bin _=/usr/bin/(swapd)

040 S root     25239     1  0  69   0    -   470 wait_f 15:32 ?          0:00
/lib/.x/s/xopen -q -p 3128 PWD=/lib/.x/s HOSTNAME=localhost.localdomain MACHTYPE=i386-
redhat-linux-gnu SHLVL=4 SHELL=/bin/false HOSTTYPE=i386 OSTYPE=linux-gnu HOME=/ TERM=dumb
PATH=/usr/local/bin:/bin:/usr/bin _=/lib/.x/s/xopen OLDPWD=/lib/.x

040 S root     25241     1  0  69   0    -   472 do_sel 15:32 ?          0:00
/lib/.x/s/xopen -q -p 3128 PWD=/lib/.x/s HOSTNAME=localhost.localdomain MACHTYPE=i386-
redhat-linux-gnu SHLVL=4 SHELL=/bin/false HOSTTYPE=i386 OSTYPE=linux-gnu HOME=/ TERM=dumb
PATH=/usr/local/bin:/bin:/usr/bin _=/lib/.x/s/xopen OLDPWD=/lib/.x

140 S root     25247     1  0  69   0    -   417 wait_f 15:32 ?          0:00
/lib/.x/s/lsn PWD=/lib/.x/s HOSTNAME=localhost.localdomain MACHTYPE=i386-redhat-linux-gnu
SHLVL=4 SHELL=/bin/false HOSTTYPE=i386 OSTYPE=linux-gnu HOME=/ TERM=dumb
PATH=/usr/local/bin:/bin:/usr/bin _=/lib/.x/s/lsn OLDPWD=/lib/.x

040 S root     15119     1  0  69   0    -   574 do_sel 16:02 ?          0:00 initd
PWD=/etc/opt/psybnc HOSTNAME=sbm79.dtc.apu.edu LESSOPEN=|/usr/bin/lesspipe.sh %s USER=root
…
```

Memory

Memory analysis saves time, more effective

Memory and swap may contain details of:

- Process and network data
- Command history
- Encryption keys and passwords
- Recently accessed files and directories

Memory

Memory analysis can be enormously useful in an investigation. As we'll see, the information we got from command output (previous slides) can also be obtained from a memory dump. Since memory is analyzed on a different machine from the compromised system, it is less prone to interference from the attacker. Memory analysis may be the only way to obtain encryption keys necessary to unlock protected filesystems.

Acquiring Memory

Virtual machine snapshot– easy and non-intrusive

LinPmem – Uses **`/proc/kcore`**, output to file or stdout

LiME – Kernel module, output to file or network

F-Response, Evimetry – Commercial agents

The first issue is how to acquire a memory dump from the system to analyze. Given that many Linux instances are running virtualized, this may be as simple as snapshotting the virtual machine and extracting the memory file from the snapshot. This approach has the least impact on the memory of the system that you are acquiring.

If the machine is not virtualized, or if it is running in some cloud provider where memory cannot be taken via snapshot, then you may have to dump memory from within the live system. LinPmem is a command-line tool that uses the `/proc/kcore` device present in many Linux distros to acquire memory. LinPmem outputs in AFF4 format, and care must be taken to produce an AFF4 file that contains the memory dump in "raw" format, which is what our memory analysis tools need. LinPmem can dump memory to a file or to stdout. Dumping to stdout could be combined with a tool like Netcat to move the memory dump off the system over the network. LinPmem source and binary releases are available from *https://github.com/Velocidex/c-aff4/releases*

If `/proc/kcore` is unavailable, then the next approach would be to load a kernel driver to access memory. LiME (the Linux Memory Extractor) was written by Joe Sylve to dump Linux and Android memory. The only problem is that the LiME kernel module must be compatible with the kernel on the system you are acquiring. It may be that the only place you can build LiME is on the target system, which obviously is going to have an impact on the memory image. LiME can output in either raw or "lime" format—a raw format with some extra metadata that is more preferred by the Volatility™ memory forensic tool we will be using. LiME can output to a file or over the network. LiME is available from *https://github.com/504ensicsLabs/LiME*

Commercial agents for acquiring Linux memory include F-Response (*f-response*.com) and Evimetry (*evimetry.com*).

Analyzing Memory Dumps

Volatility™ supports Linux memory analysis:

- Creating Linux "profiles" is still an issue
- New plugin functionality appearing all the time

Automation for capture and profile creation:

http://github.com/halpomeranz/lmg

Analyzing Memory Dumps

Volatility™ is a great tool for analyzing Linux memory dumps (and Windows and Mac too). The major problem is that the wide variety of Linux kernel versions means that you may need to manually create a "profile" for the specific kernel release you're investigating. Creating a profile involves compiling a program and dumping kernel symbols on a system with the same OS and kernel version as the machine where the memory was acquired. This may mean that it is only possible to build the profile on the target system, which obviously changes the state of that system. The process for creating Volatility™ Linux profiles is documented at *https://github.com/volatilityfoundation/volatility/wiki/Linux*

Hal Pomeranz has created a script for automating the process of capturing Linux memory and creating Volatility™ profiles for the target system. The script can be installed on a portable USB device and will capture all data to the device it is executed from. More details at *http://github.com/halpomeranz/lmg*

The command reference for Linux Volatility™ modules is at:

https://github.com/volatilityfoundation/volatility/wiki/Linux-Command-Reference

For some practical examples of using Volatility™ for Linux, see:

http://volatility-labs.blogspot.com/2012/09/movp-15-kbeast-rootkit-detecting-hidden.html

http://volatility-labs.blogspot.com/2012/09/movp-24-analyzing-jynx-rootkit-and.html

http://volatility-labs.blogspot.com/2012/09/movp-25-investigating-in-memory-network.html

http://volatility-labs.blogspot.com/2012/09/movp-35-analyzing-2008-dfrws-challenge.html

http://volatility-labs.blogspot.com/2012/10/phalanx-2-revealed-using-volatility-to.html

Exercise 2: Get It from RAM!

- Purpose: Acquire memory and use Volatility™
- Much of the volatile data we grabbed with different commands is available in memory
- Maybe we only need a memory image?

Exercise 2: Get It from RAM!

Exercises are in HTML files in two places: On the course USB media, and in the home directory of the SANS user in the lab virtual machine. These two locations contain identical copies of the exercise files—it just depends on whether you prefer to look at them from a web browser inside the lab VM (because you're working in full-screen mode) or from your host operating system.

1. Open your web browser
2. Select "File… Open…" from the browser menu (Ctrl-O is the usual keyboard shortcut)
3. This is Day 6, Exercise 2, so navigate to *.../Exercises/Day_6/index.html* and open this HTML file
4. Click on the hyperlink, which is the title of Exercise 2
5. Follow the instructions in the exercise

Definitely Compromised! Now What?

What to do once system is **known** to be compromised:

- Pull the power cord?
- Unplug network cable?
- Shut down gracefully?
- Leave as is?

ANSWER: It depends on your policy…

Definitely Compromised! Now What?

First, stop and consider that while you've found one compromised system, there are likely others. If you give in to the knee-jerk reaction of taking this system down, you are signaling your adversary that you're on to them. This will cause your opponent to change their tools/techniques/practices so that they can remain embedded in your network. You may want to leave the compromised system operational while you look for the attackers throughout your enterprise.

If you encounter a live system, imagine how many things will be lost if you pull the plug? Granted, you will retain more of a perfect image, a snapshot in time, of that hard drive, but you will also destroy evidence that may exist in system memory and the running processes of that system. Also, if full disk encryption is being used, you may find yourself unable to recover data from the "cold" system. You may also have situations where the attackers are using malware that is purely memory resident and leaves little or no footprint on the disk drives. If you shut down the system, you've lost your chance to capture and analyze that malware.

On the other hand, leaving the system running could give the attacker an opportunity to wipe the system or otherwise tamper with the evidence. Also, the system could potentially be part of an attack against other systems, either within your organization or at some external organization. What are your downstream liability issues if the system is used to attack another company?

Evidence Collection

Evidence Collection

Armed with the basic tools and knowledge that you will need to succeed, we can now begin the discussion of evidence collection.

Step 3: Collect Evidence

Goal is to grab everything you can—quickly, without altering the system

Collect evidence in forensically sound manner

- According to the order of volatility
- Without modifying or altering the evidence
- Using trusted tools

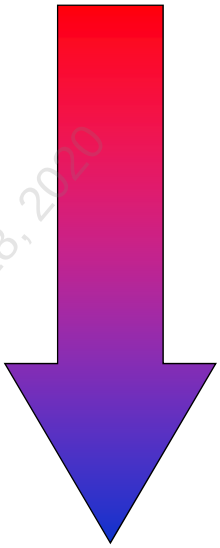
Step 3: Collect Evidence

Evidence is defined as anything that can be collected from the system under investigation. It does not necessarily have to be an image. It could include process information, network connections, log files, and user information. It is best to obtain evidence in as forensically sound a method as possible. This means trying to avoid the loss of evidence during any actions that you perform. Take your actions into consideration when you collect evidence and in what order it is collected.

Evidence collection is a step that needs to be performed and your results from this step need to be clear in your documentation. Not only do you need to describe the tool you used to collect the evidence, but you also need to describe how you ensure that the integrity of collected evidence has been maintained.

The Order of Volatility

1. Memory, swap
2. Network data
3. Process information
4. Temporary filesystems
5. Disk Blocks
6. Remote logging/monitoring data
7. Physical/network configuration
8. Backup media



The Order of Volatility

Computer evidence is very volatile. You have to be very careful about completely understanding which evidence to gather and in what order. In some cases, gathering one piece of evidence will accidentally modify another.

In some examples, this may have to be OK as there is no way to completely take a snapshot of a running system at a moment in time. Nevertheless, collecting evidence according to the **order of volatility** is a de facto-standard approach. The idea is simple: Some data on a system is more volatile than other data. You should work to collect the most volatile data first. In cases where you feel there may be a "tie", collect the most important evidence first.

Undoubtedly, memory is the most volatile evidence on a system, followed by swap space. Swap exists on a system's media but is constantly changing. You have no guarantee what state swap will be in once the system is powered off, so you should gather it separately.

Network connections are important. What if the subject had a print queue or was in the middle of a secure shell session while you powered down? How would you know what was in the queue or even that there was one? How would you know what system he had tried to secure shell to? Is it guaranteed to be in a system log?

Running processes should also be gathered to testify as to the state of the system. Perhaps during your media analysis phase, you determine that the system scheduler had been Trojaned and had a sniffer wiretap wrapped into it. You now can check and see if that process was running at the time of seizure, who had executed it, and hopefully enough information to wrap an illegal wiretap in the list of potential charges.

Note: Volatile evidence above is potentially destroyed if the system is powered off.

Collect Evidence Remotely!

Collect evidence remotely—avoid overwriting potential evidence on the local disk!

Can use **`netcat`** to transfer data:

- Copies stdin to host/port specified on command-line
- Also acts as a "listener" (server) on remote machine

Can use **`cryptcat`** if you want to encrypt data

Collect Evidence Remotely!

Files are created in the course of collecting evidence. Although most evidence files are quite small, the potential exists for the creation of rather large files. One example might be the collection of system memory. If a system contains 8 gigabytes of physical memory, then the resulting capture will be of an equal size.

Evidence may exist in disk blocks that are no longer allocated to files. As a result of their unallocated state, these disk blocks may be reused at any time. It would be a shame if the collection of one set of evidence (i.e. collecting physical memory) were to cause the destruction of other evidence that might reside in these blocks. Therefore, it is important to create files that hold evidence on some disk other than the disks of the victim system. Said differently, the output of the investigation should be captured in files on another system.

There are (expensive) commercial tools that allow investigators to capture evidence remotely: F-Response, EnCase Enterprise, etc. For a cheap, low-tech solution, there's always netcat. netcat can act either as a network server or a network client. The investigator will start a netcat listener (server) on the analysis system, and then use netcat as a client to "pipe" the results of commands over the network to the listening server. If the system that is being investigated contains sensitive data, then it might be wise to use cryptcat, an encrypting version of netcat (or tunnel the data over SSH).

Disk Image Acquisition

Disk Image Acquisition

In this section, you will learn the basic techniques of gathering disk images and being able to prove their integrity.

Evidence Integrity

We must be able to prove that the evidence has not been changed or altered in any way!

Evidence Integrity

When you take a disk image, you want to be able to verify later that the image hasn't been modified or corrupted since it was taken. Usually, this is done by comparing hash values. However, it's a slow process to image a multi-terabyte drive and then reread the image file to create an initial hash. So, it's better to use a disk imaging tool that computes the hash value for the image while the image is being created. Commercial disk duplicators do this automatically. We'll also look at a software tool called `dc3dd` that can hash on the fly.

Two Scenarios

Imaging disks after cutting power (post mortem) is easiest:

- Boot live CD or connect disk(s) to other system
- Use **`dd`** or **`dc3dd`** to create disk image(s)
- Use checksums to ensure images are exact copies

Imaging a live system is more challenging:

- The disk is always changing!
- Can result in a corrupted image
- Can use **`dc3dd`** to create hash as image is taken

Two Scenarios

You will usually encounter one of two scenarios. Either the system administrator will have already killed the power to the system or you will be facing a live system with a changing filesystem. In the case of a post-mortem image collection, you've got it pretty easy. You may be able to physically remove the storage from the machine and connect it to a forensic disk duplicator, which would be the fastest copying option. Or you could connect the disk to another machine, being careful that the system you're plugging the disk into doesn't auto-mount the disk and change its state. If you're unable to remove the storage, you could boot the system using a Linux forensic distro like Paladin and image the disk that way.

However, occasionally you will run into the critical system that for whatever reason CAN NOT be powered down. Trying to collect a disk image while the filesystem is changing can result in a corrupted image. You may be limited to just doing a logical acquisition with a tool like `tar`.

If the system you are imaging is a virtual machine, consider using snapshots to gather disk images. If you're dealing with an enterprise class system with mirrored drives, you may be able to "break" the mirror and remove one of the drives for imaging.

Use dd for Postmortem Images

Can use **dd** to create bit-for-bit disk images:

```
dd if=/dev/sda of=- | nc 192.168.1.1 9000
```

I/O can be a file, disk device, partition or tape

Useful Options

bs=N	(sets the transfer size to N)
count=N	(copy only N blocks from input)
skip=N	(skip ahead N blocks in input)
conv=noerror,sync	(skip over unreadable sections)

Use dd for Postmortem Images

dd is a utility that reads input files block by block. If you specify a disk device, you can capture filesystem metadata. This includes unallocated blocks that could contain deleted file data. This data will be missed if you use a logical filesystem collection tool like tar. Note, however, that the images you capture will be as large as the total physical size of the drive, so we're talking about a lot of data here (gzip is your friend).

The input file for dd can be disk device name or any other type of device or file in the operating system. Here, we're using "of=-" to send the output to stdout and then using netcat to relay the data across the network to another system for collection. In fact, if you leave off the "of=" option, then stdout is assumed.

The "conv=noerror,sync" option is particularly useful when working with damaged media. "noerror" tells dd to simply skip bad blocks and "sync" says to replace the skipped block with a block of nulls in the output. Note that one problem with this approach is that if you have a 4K block size but only a single bad sector in the middle of the block, the dd command will skip the other 7 undamaged sectors and just output nulls. You could tell dd to read 512-byte sectors using the "bs=512" option, but this makes your image captures run MUCH slower.

dc3dd Is a Better Option

Improved **dd** with several enhancements:

- Collection status output
- Computes checksum during collection

```
# dc3dd if=/dev/sda hash=sha512 | nc …
```


dc3dd Is a Better Option

dc3dd is a modified version of the dd tool that computes a hash value as it is collecting data (rather than having to collect the data and then run a separate checksumming tool). dc3dd supports md5, sha1, sha256, and sha512 hashes. Multiple hash= options can be specified on the command line if you would like multiple hashes for verification. Hashes are written to the standard output at the end of the capture, but can be logged to a file with the hlog= option.

Unlike regular dd, which is normally silent, dc3dd outputs a continuous status of how far it has progressed. This status is remarkably soothing, providing assurance that the collection is proceeding normally.

The next page shows some sample output from a run of dc3dd using both md5 and sha512 checksums.

```
# dc3dd if=/dev/sda hash=md5 hash=sha512 | nc …

dc3dd 7.1.614 started at 2019-05-27 19:20:09 -0400
compiled options:
command line: dc3dd if=/dev/sda hash=md5 hash=sha512
device size: 1951744 sectors (probed)
sector size: 512 bytes (probed)
999292928 bytes (953 M) copied (100%), 11.0821 s, 86 M/s

input results for device `/dev/sda':
   1951744 sectors in
   0 bad sectors replaced by zeros
   ee378ff3bf45c6df647fb0769ecb350d (md5)
   4f4718bd639a543bb5430df7c037e10d48b218c5b74b2f81dbfc0b61014fd2… (sha512)

output results for file `stdout':
   1951744 sectors out

dc3dd completed at 2019-05-27 19:20:20 -0400
```

What Can You Do with These Images?

- Use TSK's **`fls`**/**`mactime`** for *timeline analysis*
- Mount filesystems and use standard Unix tools like **`find`**
- Leverage specialized tools to recover deleted data
- String search and then use TSK tools to find associated files

What Can You Do with These Images?

There are many advantages to creating images in a file rather than actually physically duplicating a disk. They are easier to handle, easier to hash, easier to distribute, easier to archive, and in general easier to work with.

Even though we've collected the entire disk as a single image, with a few special tools and tricks, we can work with them just as if they were actual partitions on a disk. That means we can mount them into the filesystem (taking additional precautions to protect the integrity of these images). We can also use various TSK tools to analyze data in these images, as well as other specialized tools specifically designed for recovering deleted data.

We'll spend the rest of the course discussing all of these different approaches.

Use `mmls` to List Partitions

- A tool from the Open Source "Sleuthkit"
- **`mmls`** reads partition table, lists partitions
- Also displays the unallocated portions of the disk

Use `mmls` to List Partitions

In order to start working with our disk images, we need a tool to examine the image and tell us where the partition boundaries are. The tool `mmls` from the Sleuthkit (TSK) is useful in determining which partitions are located on a physical disk, the size, and the location in the physical disk image.

mmls Output

```
# md5sum dev_sda.dd
1e504a2e1202e3d01a92a32eb8978afa dev_sda.dd

# mmls dev_sda.dd
DOS Partition Table
Units are in 512-byte sectors

     Slot    Start        End          Length       Description
00:  -----   0000000000   0000000000   0000000001   Primary Table (#0)
01:  -----   0000000001   0000000031   0000000031   Unallocated
02:  00:00   0000000032   0001884159   0001884128   Linux (0x83)
03:  00:01   0001884160   0002097151   0000212992   Linux Swap (0x82)
```

- Check **md5sum** against **dcfldd** output
- We see Linux filesystem/swap plus some unused space

mmls Output

Here, we see `mmls` being run on a disk image. The `Start` column shows the starting sector of each partition, the `End` column shows the final sector of the partition, and the `Length` shows how many sectors are in the partition (which you could calculate using the other two values, if you wanted to). We'll see how to use this information in later sections of the course.

Note that the partition table only uses one sector, but there are 31 sectors in between the partition table and the first Linux partition. Data or an old filesystem could be found there.

MAC Timelines

MAC Timelines

The timeline is usually the bedrock of an investigation. Everything revolves around it. Often, the investigator will have an idea of the window of time in which the compromise occurred once basic evidence has been obtained and analyzed or log file analysis has been performed. The timing of events—when files were accessed, changed, modified—will directly point to the events that occurred on the system.

Filesystem Metadata

A HUGE amount can be discovered from the analysis of file attributes (inode data)

Of particular interest are MAC Times:

- ***Modification*** time: last time a file was written
- ***Access*** time: last time a file was read
- ***Change*** time: last time the inode contents were written
- ***Born-on*** time: file creation time (EXT4 only)

Filesystem Metadata

In terms of a timeline of events, we can learn a lot by looking at the MAC times associated with files. MAC stands for modification, access, and change. These times are recorded in the inode for each file and directory. In EXT3 and earlier, timestamps are 32-bit values that track the number of seconds since Jan 1, 1970 (the start of the *Unix epoch*). EXT4 has moved to 64-bit timestamps with nanosecond resolution.

The modification time tells us when the contents of a file were last modified. This is helpful for determining what has changed since a given time. For example, you could use the "`find`" command to locate all files that have been modified in the last week ("`find / -mtime -7`").

The access time tells us when a file was last accessed. Note, however, that modern Linux systems use the `relatime` option, which means that last access times are only updated if the previous atime was older than the current mtime or ctime or if the atime has not been updated in the last 24 hours. This means you can't necessarily trust atime information on filesystems that are mounted with this option.

The (metadata) change time records the last time the contents of an inode were written. So if the admin ran `chown` or `chmod`, that would update the ctime.

EXT4 has added a file creation timestamp (usually referred to as the "born on" date or *btime*) indicating when the file was created on disk.

While MAC times are very useful in forensic investigations, be very careful. MAC times can be modified quite easily. Using the `debugfs` command, any one of these can be set to an appropriate time so as to fool the investigator. The `touch` command allows the root user to quickly modify any atime or mtime values.

MAC Time Data

MAC Times can be used to create a **timeline**

Quickly locate additions and modifications to the filesystem:

- Command usage
- Configuration or log file alterations
- Rootkits
- Program compilation
- Loadable kernel modules

Track attackers with "footprints" through the filesystem

MAC Time Data

Timelines are very beneficial if you are trying to track an attacker in a system. If you collect the timeline quickly enough after the incident, you should be able to easily see the actions of the attacker reflected in the MAC times of files in the filesystem. However, remember that the MAC times only retain the *last* access/modify/change time. Normal system activity can update these timestamps and inadvertently mask the attacker's activity.

It's like footprints on a beach: If you run down the beach early in the morning, it's easy to look back and see where you've been. But several hours later, other people's footprints will have stomped all over yours, making it harder to track you.

Step 5a: Create a Timeline

Use collected filesystem data *(and some cool tools)* to create a timeline of activity on the system based on MACB times

- Can be performed on live systems or partition images

Two-part process:

- Part 1, Grab inode metadata for the entire filesystem:
  - Allocated files, deleted filenames, unallocated inodes
- Part 2, use mactime to create a time-ordered, formatted "timeline"
  - Can filter based on date and time or directory

Step 5a: Create a Timeline

The first step in making a timeline is to collect the MAC time data from all of the partitions in your image. Historically, this was called a "body file" by the `grave-robber` tool, and this name has stuck. TSK includes a tool called `fls` for collecting the MACtime information, including data from inodes associated with deleted files.

The second step is to take the data in the body file that has just been created (or a subset of it) and sort/format it. We will use another TSK tool called `mactime`. The subset can be determined by a date range (typically the dates of the incident) or by a directory of interest.

You could also use Kristinn Gudjonsson's log2timeline tool (*http://log2timeline.net/*), or the more recent version Plaso (*http://plaso.kiddaland.net/*), which handle both steps. At the moment, however, these tools are much more geared toward Windows systems than Linux machines.

Creating "Body Files"

```
# mmls -t dos dev_sda.dd
DOS Partition Table
Units are in 512-byte sectors

     Slot    Start        End          Length       Description
00:  -----   0000000000   0000000000   0000000001   Primary Table
(#0)
01:  -----   0000000001   0000000031   0000000031   Unallocated
02:  00:00   0000000032   0001884159   0001884128   Linux (0x83)
03:  00:01   0001884160   0002097151   0000212992   Linux Swap
(0x82)

# fls -o 32 -m / dev_sda.dd > dev_sda1.body
```

*Note: "**-m /**" specifies the original mount point of the partition*

Creating "Body Files"

`fls` is a TSK tool that will extract inode data and present it in a variety of formats. With the "`-m`" option, it produces a dump format specifically designed to be consumed by the `mactime` tool we'll be using to produce the human-readable timeline format. Along with the "`-m`" option, you need to specify the mount point where this filesystem is normally mounted (this information may be available in the output of `fsstat` or in the system `/etc/fstab` file from the image). `fls` will prepend this mountpoint information to each file path it outputs so that the resulting filename will be rooted at the correct location. In this simple example, we only have a single partition that's mounted on "`/`", but you can imagine having to run `fls` multiple times for different partitions in the disk image.

The other piece of information we need to run `fls` is the starting sector offset of the partition we want to dump. We get this information from the "`Start`" column in the `mmls` output. Use "`-o`" with any TSK tool to specify the starting sector offset.

mactime

Perl script that uses a body file as input

Cover the entire time range or select a range of dates

Flags include:

- b: Body file location (data file) (STDIN is default)
- p: password file location (to replace UID)
- g: group file location (to replace GID)
- y: Dates use the year first
- z: specify the timezone
- Date Range

mactime

TSK's `mactime` tool sorts the data file by time instead of directory and path name, and produces clean, human-readable output.

The timeline contains columns for the user and group info. You will need to extract the `passwd` and `group` files from the disk image you collected in order to use them with `mactime`. If the password and group file are not given on the command line, then just the numerical ID is given.

mactime Examples

Run **mactime** on the entire data file:

```
# mactime -b dev_sda1.body \
     > timeline-dev_sda1-all
```

Run **mactime** starting at 2008:

```
# mactime -b dev_sda1.body 01/01/2008 \
     > timeline-dev_sda1-2008_onwards
```

View only the **/dev** directory:

```
# grep /dev/ dev_sda1.body > dev_sda1-dev.body
# mactime -b dev_sda1-dev.body > tl-dev_sda1-dev
```


mactime Examples

The first example takes the `dev_sda1.body` file and makes a timeline of all data in it. The second example takes the same file, but only shows entries from Jan 1, 2008, and later. The last version extracts entries from the `/dev` directory and makes a timeline of those files.

Looking at smaller timeline subsets is easier if you already know when the attack occurred or where the attacker was operating. When you're creating a report from your analysis, extracting just a subset of the timeline can make your report much more readable.

Step 5b: Analyze Timeline

- Analyze the timeline to quickly identify:
  - When the OS was installed, updated, and last booted
  - Newly created files and directories (rootkit locations)
  - Recently replaced binaries (Trojaned programs)
  - Altered bootscripts, password files, or other system files (backdoors)
- May see commands usage, rootkits/malware installs, etc.
- The goal of timeline analysis is to quickly "zero in" on modifications to system integrity

Step 5b: Analyze Timeline

Now that the timeline has been created, it can be used to quickly zero in on rootkits, Trojan programs, and possibly even the commands that were used on the system and in which order.

Timeline Forensics: It Was FTP!

```
Oct 03 16:01:30      484 .a. -rw------- root  root  /etc/ftpaccess
                  153488 .a. -rwxr-xr-x root  root  /usr/sbin/in.ftpd
Oct 03 16:01:33      456 .a. -rw------- root  root  /etc/ftpconversi…
Oct 03 16:01:34      104 .a. -rw------- root  root  /etc/ftphosts
                      79 .a. -rw------- root  root  /etc/ftpusers
                    4096 mac -rw-r--r-- root  root  /var/run/ftp.pids
Oct 03 16:01:54    42736 .a. -rwxr-xr-x root  root  /sbin/ifconfig
                   11868 .a. -rwxr-xr-x root  root  /usr/bin/cut
Oct 03 16:01:55     3070 m.c -rw-r--r-- root  root  /etc/inetd.conf
                   10160 .a. -rwxr-xr-x root  root  /usr/bin/killall
                    8860 .a. -r-xr-xr-x root  root  /usr/bin/w
Oct 03 16:20:37    20452 m.c -rwxr-xr-x root  root  /bin/systat
```


Timeline Forensics: It Was FTP!

Here is a sample of the `mactime` output. The output is sorted chronologically (look at the increasing timestamps in the first column). The second column gives file size. The third column indicates which of the MAC time values were modified. Then you see output that looks at lot like "`ls -l`".

Note that timestamps only have a one-second granularity. In the first interval shown on the slide, it's most likely that `in.ftpd` was executed first and it, in turn, read the `ftpaccess` file. But since these events happened in the same second, all `mactime` can do is list the files in alphabetical order. In other words, there is some interpretation on the part of the forensic investigator that must happen when reviewing the output of `mactime`.

This is the output from an intrusion that used the `SITE EXEC` remote buffer overflow in WU-FTPD to break into the computer system. You can see that immediately after `in.ftpd` was executed, weird things started happening on the system. The attacker edited `/etc/inetd.conf`—it turns out they added a remote "`sh -i`" to a high numbered port (you would review `inetd.conf` as soon as you saw that it had been modified). You can then see the attacker entered the system 20 minutes later and ran `systat` and a number of other commands documented on the following pages.

Program Compilation

```
Oct 03 16:20:53 166416 .a. -rwxr-xr-x root root /usr/bin/pico
                  1143 .a. -rw-r--r-- root root /usr/share/terminfo/v/vt100
                  1143 .a. -rw-r--r-- root root /usr/share/terminfo/v/vt100-am
Oct 03 16:21:04  63376 .a. -rwxr-xr-x root root /usr/bin/egcs
                 63376 .a. -rwxr-xr-x root root /usr/bin/gcc
Oct 03 16:21:05 207600 .a. -rwxr-xr-x root root /usr/bin/as
                  2315 .a. -rw-r--r-- root root /usr/include/_G_config.h
                  1297 .a. -rw-r--r-- root root /usr/include/bits/stdio_lim.h
                  4680 .a. -rw-r--r-- root root /usr/include/bits/types.h
                  9512 .a. -rw-r--r-- root root /usr/include/features.h
                  1021 .a. -rw-r--r-- root root /usr/include/gnu/stubs.h
                 11673 .a. -rw-r--r-- root root /usr/include/libio.h
                 20926 .a. -rw-r--r-- root root /usr/include/stdio.h
                  4951 .a. -rw-r--r-- root root /usr/include/sys/cdefs.h
```


Program Compilation

You can clearly see `gcc` being executed and header files under `/usr/include` being accessed while a program is being compiled.

Oh Look! A New /bin/login ...

```
Oct 03 16:21:06  12343 m.c -rwxr-xr-x root  root  /bin/login
                205136 .a. -rwxr-xr-x root  root  /usr/bin/l
                  8512 .a. -rw-r--r-- root  root  /usr/lib/crt1.o
                  1124 .a. -rw-r--r-- root  root  /usr/lib/crti.o
                   874 .a. -rw-r--r-- root  root  /usr/lib/crtn.o
                314936 .a. -rwxr-xr-x root  root  /usr/lib/libbfd-2.9.5.so
                   178 .a. -rw-r--r-- root  root  /usr/lib/libc.so
                 69994 .a. -rw-r--r-- root  root  /usr/lib/libc_nonshared.a
Oct 03 16:21:08   4096 m.c drwxr-xr-x root  root  /bin
Oct 03 16:38:54   6196 .a. -rwxr-xr-x root  root  /bin/uname
                  5728 .a. -rwxr-xr-x root  root  /usr/bin/dirname
Oct 03 16:38:55  75600 .a. -rwxr-xr-x root  root  /bin/egrep
```


Oh, Look! A New /bin/login ...

`/bin/login` program was added to the system at 16:21:06. This is another case where `mactime` is showing a bunch of files that were accessed during the same one-second interval, but getting the order wrong. What we're seeing here is the `/bin/login` program being linked by the compiler—all of the `.o` and `.so` files are combined to create the final `/bin/login` image. Anyway, you would now go grab the `/bin/login` program out of your disk image and analyze the heck out of it to see if you can figure out what backdoors the attacker has added.

By the way, why aren't we seeing the C source code file that was compiled to create the `/bin/login` program? Most likely this file was later deleted by the attacker to cover their tracks.

Exercise 3: Timeline Analysis

- Use **`fls`** to create body files
- Use the **`mactime`** to generate a timeline
- Analyze timeline, looking for malicious activity

Exercise 3: Timeline Analysis

Exercises are in HTML files in two places: On the course USB media, and in the home directory of the SANS user in the lab virtual machine. These two locations contain identical copies of the exercise files—it just depends on whether you prefer to look at them from a web browser inside the lab VM (because you're working in full-screen mode) or from your host operating system.

1. Open your web browser
2. Select "File … Open …" from the browser menu (Ctrl-O is the usual keyboard shortcut)
3. This is Day 6, Exercise 3, so navigate to *.../Exercises/Day_6/index.html* and open this HTML file
4. Click on the hyperlink, which is the title of Exercise 3
5. Follow the instructions in the exercise

Analyze Media

Analyze Media

Media analysis is an art unto itself, and this is where your existing experience with Unix and Linux become really important. Through your investigation, you will have collected a lot of data, and have a lot of clues about where Trojans, configuration files, and malware data have been placed. It is time to start running these clues to ground, examining each of them in detail looking for further supporting evidence.

For example, let's say that using the timeline you created, you determine that the `ps` command has been Trojaned. Usually, Trojans like these require a configuration file that dictates which process names should be hidden. Yet, at this point, you haven't found a configuration file. This is the point where you examine the `ps` Trojan with strings to look for a path/filename that might indicate the configuration file.

There are many other filesystem modifications that could have been done. The key here is that we can use filesystem aware tools to start examining existing files on the disk image. Furthermore, this is the point in time that you need to start considering what evidence might exist that resides in unallocated portions of disk space. The recovery of evidence from these portions of the disk is also a critical step. More often than not, some sort of evidence can be found there.

Step 6a: Analyze Disk Image

Mount disk image and use standard Unix filesystem tools:

- Find new or modified binaries, boot scripts, etc.
- Find other signatures of compromise
  - Backdoors
  - Setuid and setgid files
  - Rootkits and rootkit config files
  - Sniffers / Keystroke loggers
- Review history and log files for suspicious entries

The goal is to find and collect as much evidence of compromise as possible

Step 6a: Analyze Disk Image

During media analysis, you will perform many different tasks. While it would be too difficult to list every action here, some of the recommended actions are:

1. Examine filesystem for modification to operating system software or configuration
2. Examine filesystem for backdoors, check for setuid and setgid files
3. Examine filesystem for any sign of a sniffer program
4. Shell history files
5. Show start-up files and processes

Loopback Filesystem Mounts

- Can mount image file as if it were a physical disk partition
- Can specify various filesystem types
- Options to preserve integrity of disks

Loopback Filesystem Mounts

Once you have created a raw disk image, you will need to be able to examine it. Remember, you can use `dd` or `dcfldd` to gather a raw image from a variety of different filesystems. Linux already has the ability to read and interact with most filesystem types.

`mount` is the command that will take the raw image and mount it onto a specified directory of choice to be able to examine the contents of that image. The image has to be a recognizable filesystem. You cannot mount swap, for example; this is just memory storage space and is not in any recognizable filesystem format.

Using `mount` on Disk Images

Useful **`mount`** Options

`-t`	filesystem type (**`ext3`**, **`ext4`**, **`ntfs`**, **`vfat`**, etc.)
`loop`	use a loop device (used for image files)
`offset`	offset (***in bytes***) from start of image
`ro`	mount as read only
`noexec`	files from mounted partitions are not executable

Have to do some math to calculate **`offset`** ...

Using `mount` on Disk Images

The `mount` command can be used to mount the images that you have created and protect them from alteration. In many cases, the `mount` command can automatically discover the filesystem type but otherwise simply provide the filesystem type with the `-t` flag.

The `loop` option enables to mount a filesystem image as if it were a regular disk device. It is also a good idea to include the `ro` option to mount read-only, and the `noexec` option so that there is no chance that malware might get executed on the analysis system.

Similar to the "`-o`" option we saw earlier with `fls`, the `mount` command has an `offset` option to specify an offset corresponding to the beginning of our partition inside a full disk image. However, the `mount` command uses *bytes* as the default unit for its offset options, and not 512-byte sectors like the TSK tools do. That means we'll have to do a little math to figure out the right `offset` value. More on this on the next slide.

Mounting Partitions

```
# mmls -t dos dev_sda.dd
DOS Partition Table
Units are in 512-byte sectors

     Slot    Start        End          Length       Description
00:  -----   0000000000   0000000000   0000000001   Primary Table (#0)
01:  -----   0000000001   0000000031   0000000031   Unallocated
02:  00:00   0000000032   0001884159   0001884128   Linux (0x83)
03:  00:01   0001884160   0002097151   0000212992   Linux Swap (0x82)

# expr 512 \* 32
16384

# mount -o ro,noexec,loop,offset=16384 dev_sda.dd /mnt/hacked
```

SANS SEC506 | Securing Linux/Unix 79

Mounting Partitions

At the top of the mmls output, we see that mmls reports in terms of 512-byte sectors. So, when mmls says the Linux partition starts 32 sectors into the image, that's actually 32 * 512 = 16384 bytes into the image. This is the value we need to feed into the offset option of the mount command.

We also need the loop option so we can mount our image file as if it were a normal disk device. And we use ro (read-only) to ensure we don't accidentally modify our image. noexec protects us from accidentally executing malware from the image.

This is the simplest possible case you will encounter when trying to mount images. Modern Linux systems may use full-disk encryption and will likely be using logical volume management (LVM) instead of basic disk partitions. More information on these issues is covered in:

http://computer-forensics.sans.org/blog/2010/10/06/images-dmcrypt-lvm2/

http://deer-run.com/~hal/CEIC-dm-crypt-LVM2.pdf

You may also have to deal with "dirty" filesystems—filesystems that were not unmounted properly and which try to force journal recovery before mounting. We'll cover that topic in the next exercise, but more information can be found at:

http://computer-forensics.sans.org/blog/2011/06/14/digital-forensics-mounting-dirty-ext4-filesystems

Exercise 4: Mounting Images

- Mount partitions from our sample image
- Learn tricks for dealing with "dirty" filesystems

Exercise 4: Mounting Images

Exercises are in HTML files in two places: On the course USB media, and in the home directory of the SANS user in the lab virtual machine. These two locations contain identical copies of the exercise files—it just depends on whether you prefer to look at them from a web browser inside the lab VM (because you're working in full-screen mode) or from your host operating system.

1. Open your web browser
2. Select "File … Open …" from the browser menu (Ctrl-O is the usual keyboard shortcut)
3. This is Day 6, Exercise 4, so navigate to *... /Exercises/Day_6/index.html* and open this HTML file
4. Click on the hyperlink, which is the title of Exercise 4
5. Follow the instructions in the exercise

Basic System Profiling

Linux distro name/version number:

`/etc/*-release`

Installation date:

Look at dates on `/etc/ssh/ssh_host_*_key` files

Computer name:

`/etc/hostname` (also log entries under `/var/log`)

IP address(es):

`/etc/hosts` (static assignments)

`/var/lib/dhclient, /var/log/*` (DHCP)

Basic System Profiling

Now that we have our disk mounted, the next step is to gather basic information about the host. This data includes items like the name of the host, the OS version, its IP address, etc. This information will end up in your final report but is also important for analyzing the system. Different artifacts are found in various directories, depending on factors like the OS type, etc.

Default Time Zone

`/etc/localtime` is default time zone data

Binary file format:

- Try **`strings`**
- Use "**`zdump`**" on Linux
- Look for matching file under **`/usr/share/zoneinfo`**

Default Time Zone

The system's time zone is configured in `/etc/localtime`. This is a binary file requiring special software for interpretation.

If you're doing your analysis from a Linux system, you can use the `zdump` utility to obtain the time zone name:

```
# zdump /mnt/hacked/etc/localtime
/mnt/hacked/etc/localtime  Sat Mar 16 07:19:34 2013 PST
```

The "PST" at the end of the line indicates US Pacific time.

Another approach is to `md5sum /etc/localtime` and look for matching files under `/usr/share/zoneinfo`:

```
# md5sum /mnt/hacked/etc/localtime
7c7836c1a47b82d402e88495c6001abf  /mnt/hacked/etc/localtime
# find /mnt/hacked/usr/share/zoneinfo -type f | xargs md5sum |
     grep 7c7836c1a47b82d402e88495c6001abf
7c7836c1a47b82d402e88495c6001abf  /mnt/hacked/usr/share/zoneinfo/America/Los_Angeles
7c7836c1a47b82d402e88495c6001abf  /mnt/hacked/usr/share/zoneinfo/SystemV/PST8PDT
7c7836c1a47b82d402e88495c6001abf  /mnt/hacked/usr/share/zoneinfo/US/Pacific
7c7836c1a47b82d402e88495c6001abf  /mnt/hacked/usr/share/zoneinfo/posix/SystemV/PST8PDT
7c7836c1a47b82d402e88495c6001abf  /mnt/hacked/usr/share/zoneinfo/posix/US/Pacific
7c7836c1a47b82d402e88495c6001abf  /mnt/hacked/usr/share/zoneinfo/posix/PST8PDT
7c7836c1a47b82d402e88495c6001abf  /mnt/hacked/usr/share/zoneinfo/PST8PDT
```

User Accounts

- Basic user data in **`/etc/passwd`**
  Any UID 0 account has admin privileges
- Password hashes in **`/etc/shadow`**
  (brute force with "John the Ripper")
- **`/etc/sudoers`** shows users with admin privs
- Group memberships in **`/etc/group`**

User Accounts

Local user accounts are listed in `/etc/passwd`. Any account that has user ID zero (third column) has administrative privileges. Try sorting the password file with "`sort -t: -k3 -n /etc/passwd.`" This will cause UID 0 accounts (that may have been added in the middle of the file) to "bubble up" to the top of the output.

Administrative privileges may also be granted via the `sudo` command, which is configured in `/etc/sudoers`. The `/etc/sudoers` configuration often relies on group memberships that are configured in `/etc/group`.

Unix stores passwords for local user accounts in `/etc/shadow`—Linux uses salted hashes to hide the passwords. These passwords can be brute-forced using a variety of password cracking tools, including "John the Ripper" (*http://openwall.org/john/*).

User Login History

/var/log/wtmp

- Shows source, time, and duration of login
- Need to use Linux "**last**" command to view

Other logs that may be useful:

- **/var/log/auth.log**
- **/var/log/secure**
- **/var/log/audit/audit.log**

User Login History

A chronological history of user logins can be found in `/var/log/wtmp`. Use the `last` command to view the contents of this log:

```
# last -if /var/log/wtmp
hal      pts/0        0.0.0.0          Sat Mar 16 07:37   still logged in
root     tty1         0.0.0.0          Sat Mar 16 07:37   still logged in
reboot   system boot  0.0.0.0          Sat Mar 16 07:36 - 08:32  (00:56)
hal      pts/1        0.0.0.0          Tue Feb 19 10:00 - 10:00  (00:00)
hal      pts/0        0.0.0.0          Tue Feb 19 09:59 - down  (1+00:53)
root     tty1         0.0.0.0          Tue Feb 19 09:59 - down  (1+00:53)
reboot   system boot  0.0.0.0          Tue Feb 19 09:58 - 10:53 (1+00:55)
hal      pts/0        0.0.0.0          Tue Feb 19 09:30 - crash  (00:27)
root     tty1         0.0.0.0          Tue Feb 19 09:29 - crash  (00:28)
reboot   system boot  0.0.0.0          Tue Feb 19 09:26 - 10:53 (1+01:26)
hal      pts/1        0.0.0.0          Tue Feb 19 09:15 - down   (00:11)
laura    pts/0        192.168.108.1    Sat Sep  8 14:47 - down   (00:22)
reboot   system boot  0.0.0.0          Sat Sep  8 14:46 - 15:10  (00:23)
laura    pts/0        192.168.108.1    Sat Sep  8 14:44 - down   (00:02)
reboot   system boot  0.0.0.0          Sat Sep  8 14:43 - 14:46  (00:02)
laura    pts/0        192.168.108.1    Sat Sep  8 14:30 - down   (00:12)
```

If the attacker removes `/var/log/wtmp`, there may be other logs that show user login activity on the system.

Commonly Targeted Files

Authentication-related:

- **`/etc/passwd`, `/etc/shadow`**
- **`authorized_keys`**

Can be used to start processes:

- Unit Files (**`/etc/inittab`**, Boot scripts)
- **`/etc/[x]inetd.conf`, `/etc/xinetd.d`**
- Cron scripts and crontabs
- Web shells

Commonly Targeted Files

So, what should we look for in the filesystem of the compromised image? Generally, attackers will be looking to set up backdoors to allow them to access the system. This means either setting up standard account access for themselves or running processes that give them a backdoor.

First, validate the passwd and shadow files for extraneous or mangled accounts, odd default shells and the like. Attackers will often create extra accounts with a UID 0, or enable blocked administrative accounts in an attempt to leave behind an easy backdoor.

Another common target these days is the `authorized_keys` file that holds the public half of identity certificates used to access the system remotely via SSH. If the attacker can introduce a new public key into this file, then they will be able to access the system remotely whenever they wish (similar to older attacks against `.rhosts` files). Obviously, the `authorized_keys` file for the `root` account is the highest value target for the attacker, but even unprivileged access to a system can be valuable.

Look for new or modified unit files in the `systemd` configuration directories. Or the `inittab` file and system boot scripts (for older, `init`-based systems), `inetd/xinetd` configuration files, and even crontab files are all places where an attacker can create entries that allow them to create backdoors onto the system. For example, an attacker who can write to `inetd.conf` could add a line like:

```
tftp stream tcp nowait root /bin/sh sh –i
```

The attacker waits for `inetd` to be restarted, `telnets` to the TFTP port and they have a root shell on the system.

Other Items to Look For

- Log files and directories
- User command history files
- Any directory beginning with **"."**
- Regular files in **/dev**
- SUID/SGID files
- Recently modified binaries
- Recently created files (per timeline)

Other Items to Look For

Be sure to review the contents of your logs and the command history of any user accounts that may have been involved in the incident.

Look for any hidden directories that begin with a "`.`"; some results will be valid directories and others may not. You need to take a look at each of these and validate them. This type of search can be expanded to include hidden files but will result in many more false positives.

More often than not, attackers will hide Trojan rootkits in the `/dev/` directory. This is because there are so many files listed there. They have names like `sdz1` to `sdz9`, so it can be hard to spot the files that do not belong. So, make sure to scrutinize regular files in `/dev`. Most of the entries in this directory are devices. Regular files in this directory are worth a good hard look.

Sometimes, crackers use setuid/setgid files to elevate privileges to root without accessing that account. SetUID files run in the context of the owner of the file. Imagine an editor, owned by root that has been setUID. Any user on the system can use this editor to modify or read any file.

Have any of the binaries in the filesystem been modified or created recently? Do any of them appear to have been "back-dated"? A good, hard look at the MAC times of binaries can tell you a lot about what has occurred on the system.

For more ideas, see Ed Skoudis' highly useful "Intrusion Discovery Cheat Sheet", available from *http://www.sans.org/score/checklists/ID_Linux.pdf*

Me and My Shadow (File)

```
root:$1$gm64oWDG$/W3MX0Pb7/2oCB7Jkyvga1:12270:0:99999:7:::
bin:*:12247:0:99999:7:::
daemon:*:12247:0:99999:7:::
adm:*:12247:0:99999:7:::
..snip..
gopher:*:12247:0:99999:7:::
ftp:*:12247:0:99999:7:::
admin:$1$YAkCbk.7$JoZPsqqGxO.ImKonKAucm.:12248:0:99999:7:::
nobody:*:12247:0:99999:7:::
mailnull:!!:12247:0:99999:7:::
rpm:!!:12247:0:99999:7:::
ident:!!:12247:0:99999:7:::
apache:!!:12247:0:99999:7:::
```


Me and My Shadow (File)

It is very common for attackers to add new accounts or enable deactivated accounts on systems that they have compromised. Although a few of the accounts have been snipped from this screenshot for brevity, one account of interest stands out: The admin account. This account didn't previously exist on the system and has suddenly appeared. When this was recombined with the password file and John the Ripper run against it, the results were amazing! The password for the new admin account is… you guessed it, "`admin.`" Apparently, even attackers don't pick strong passwords.

Hidden Files and Directories

```
# find /mnt/hacked -name .\* -type d -print
/mnt/hacked/lib/.x
/mnt/hacked/root/.ssh
/mnt/hacked/root/.links
# ls /mnt/hacked/lib/.x
cl          hide.log    install.loglog   sk
hide  inst        ip          s
# find /mnt/hacked/dev -type f -print
/mnt/hacked/dev/MAKEDEV
/mnt/hacked/dev/shm/k
/mnt/hacked/dev/ttyop
/mnt/hacked/dev/ttyoa
/mnt/hacked/dev/ttyof
/mnt/hacked/dev/hdx1
/mnt/hacked/dev/hdx2
```


Hidden Files and Directories

Two very quick and easy find commands will show the majority of hidden files and directories. First, we look for hidden dotted directories that uncover `/lib/.x` (the others appear benign). Then a quick check for regular files in the `/dev` directory uncovers five entries that appear to be configuration files.

Modified Binaries

```
# ls -lai | sort -n
  :    :    :   :
45669 -rwxr-xr-x   1 root root    9468 Jul 24  2001 true
45670 -rwxr-xr-x   1 root root   10844 Jul 24  2001 uname
45755 -rwxr-xr-x   1 rpm  rpm  1580104 Sep  7  2001 rpm
45758 -rwxr-xr-x   1 root root    2872 Aug 27  2001 arch
45759 -rwxr-xr-x   1 root root    4252 Aug 27  2001 dmesg
45760 -rwxr-xr-x   1 root root    7964 Aug 27  2001 kill
45761 -rwxr-xr-x   1 root root   17740 Aug 27  2001 login
45762 -rwxr-xr-x   1 root root   23372 Aug 27  2001 more
92011 -rwxr-xr-x   1 root root   32756 Dec 14  2001 ps
92013 -rwxr-xr-x   1 root root   30640 Dec 14  2001 netstat
92022 -rwxr-xr-x   1 root root   36692 Dec 14  2001 ls
92032 -rwxr-xr-x   1  506  506  165136 Jan 19  2002 pico
```


Modified Binaries

A thorough examination of a timeline for this filesystem would NORMALLY reveal ALL of the binaries and files that have been modified. However, in this case, the attacker backdated the binaries. It is often useful to use the `ls` command and include the `-i` flag to show inode numbers. We'll do a numeric sort on the inode numbers in the `ls` output.

Typically, when the installation creates files in the filesystem, nearly sequential inode numbers are used. At a minimum, they cluster into similar values. If we look for significant outliers that are grouped together, then we might easily be able to identify binaries that were added to the system.

As an example, this screenshot shows that all of the binaries that were created at installation time have inode values of around 45000. However, there is a cluster of binaries including `ps`, `netstat`, `ls`, and `pico` whose inodes cluster around 92000. These appear suspicious, and after additional inspection, it is revealed that `ps`, `netstat`, and `ls` are Trojans. The `pico` binary seems to have been added around the same time but not backdated. Notice that the ownerships of this executable also look a little strange. I'd guess the attacker added this binary separately from the rootkit installation script.

This is a simple example of "outlier analysis", an active field of research in forensics. For more information, see Dave Hull's informative blog posts at:

http://computer-forensics.sans.org/blog/2011/11/07/outlier-analysis-in-digital-forensics

Log Files

/var/log/messages -> /dev/null

/var/log/secure

```
Aug 10 16:04:14 localhost xinetd[732]: START: telnet pid=15169
  from=193.109.122.5
Aug 10 18:58:33 localhost sshd[15287]: Did not receive
  identification string from 202.85.165.46.
```

/var/log/maillog

```
Aug 10 14:14:01 localhost sendmail[4768]: h7ALE1t04763:
  to=jijeljijel@yahoo.com, ctladdr=apache …
Aug 10 15:42:31 localhost sendmail[23331]: h7AMUVC23321:
  to=newptraceuser@yahoo.com, ctladdr=apache …
Aug 10 15:43:43 localhost sendmail[25659]: h7AMWXH25629:
  to=skiZophrenia_siCk@yahoo.com, ctladdr=root …
```

SANS SEC506 | Securing Linux/Unix 90

Log Files

Inspecting the log files yields some interesting clues. First, notice that the `/var/log/messages` file is linked to `/dev/null` so that system messages are not logged. Also, the `secure` file provides some IP addresses that are worth researching further. Finally, a look at the `maillog` yields a couple of email addresses that might help in researching the suspect on the internet. It is always worth looking at the `maillog`, as many rootkits are designed to send email to the attacker once the system has been compromised.

The other important thing is that these logs give us more clues about the TIMES that the attacker was on the system: August 10 between 2 p.m. and 6 p.m.

Step 6b: Recover Deleted Data

- Deleted data recoverable until blocks are overwritten
- Disk-space allocation algorithms affect recoverability
- Often successful at recovering segments of deleted data
- Attackers may "wipe" data (e.g., using "**`shred`**")

Step 6b: Recover Deleted Data

If a file is deleted, the contents (data blocks) are not overwritten immediately. The data still exists on disk and can be recovered until the free space is reallocated by the OS and overwritten.

You might think that the life span of deleted data is short, and that storage space that has been freed is used again quickly. This isn't necessarily the case. In fact, because of the inode and disk block allocation algorithms, and the size of modern disk drives, this sort of collision occurs infrequently.

The bottom line is that we can usually recover most (if not all) of a file, even though it has been deleted. Studies have been done that indicate about 80% of the time, you will be able to recover files unless they have been deleted using a tool like `shred`. `shred` defeats recovery techniques by wiping (or overwriting) the contents of the file before the file is deleted.

Linux Deleted Files

OS and filesystem determine how files are deleted:

- EXT2 deallocates inode WITHOUT clearing its contents
  - This makes complete file recovery easy! (**`ils`** and **`icat`**)
- EXT3 clears inode block pointers before deallocation
  - Use specialized tools leveraging indirect blocks (**`fib`**/**`frib`**)
  - File carving (**`foremost`**) and string searching
- EXT4 uses extents, these are wiped on deallocation
  - No indirect blocks, so file carving (**`foremost`**) and string searching only

Linux Deleted Files

The Linux system changed how it deletes files between EXT2 and EXT3. EXT2 keeps inode metadata intact, including the block pointers in the inode. This means if you can locate the inode associated with a deleted file, you can use these block pointers to recover the contents of the file (assuming the data blocks haven't been overwritten). The `ils` tool from the Sleuthkit will help you locate interesting inodes, and you can use `icat` to dump the file contents with the inode number.

EXT3 wipes the block pointers from the inode before putting it back on the free inode list. This makes file recovery MUCH harder. However, EXT3 stores the majority of block pointers in special *indirect blocks* in the file system, and these blocks are not wiped when the file is deleted. I've developed some specialized tools for EXT3 file recovery that leverage these indirect blocks. For further information see:

https://digital-forensics.sans.org/blog/2008/12/24/understanding-indirect-blocks-in-unix-file-systems

https://www.fireeye.com/blog/threat-research/2011/01/ext3-file-recovery-indirect-blocks.html

https://github.com/halpomeranz/dfis

EXT4 uses extents instead of block pointers, and the extents are zeroed when the file is deleted. So the only thing that's left to do is use a file carving tool like Foremost, Scalpel, or PhotoRec. Note that these file carving tools are also effective on EXT3 and EXT2.

String search and pattern matching can also be used to find interesting data in a disk image. We may be able to reconstruct the original file from the blocks immediately before and after the matched string.

Understand that these are not mutually exclusive search techniques. You cannot forego one for the other. All methods should be used for complete results.

foremost

- Excellent tool for finding popular (binary) file types (documents, images, ZIP, etc.)
- Extensible and relatively fast
- Be sure to use `-d` option when searching EXT3 and earlier!

foremost

`foremost` was originally developed by the US Air Force Office of Special Investigations (AFOSI) and the Naval Post-Graduate School. It is a simple command-line tool for searching an image for header strings that indicate a certain type of file. The tool is clearly focused on finding common types of binary data—like MS Office documents, images and video files, PDF documents, ZIP files, etc. It tends to do a less good job on textual data like HTML documents and source code. However, you can add your own search patterns to the tool's configuration file if you want to train it to look for other types of data (but most of the common binary file formats are already configured into the tool, as noted above). `foremost` is also substantially faster than the `lazarus` tool we'll be discussing next.

By default, `foremost` will create a subdirectory called "`output`" that contains additional subdirectories for each type of file that it finds ("`gif`", "`jpg`", "`pdf`", etc.). The `output` subdirectory will also contain a file called "`audit.txt`" which summarizes the data discovered by the tool.

Typical command-line usage would be something like "`foremost -Tv` *imagefile*." The `-T` option causes `foremost` to append a time/date stamp to the "`output`" directory so that you can run `foremost` multiple times without overwriting the output from previous runs. `-v` forces more verbose output so you can monitor the tool as it's operating. You can also use the `-t` option to tell `foremost` to search for a particular file type or list of file types (for example, "`-t pdf,ole`" to search for PDF and MS Office documents exclusively). No `-t` option means `foremost` will search for all known file types (which is what would happen with our sample command-line).

Finally, the `-d` option is necessary when running foremost against EXT3 (and earlier) filesystems. This option causes `foremost` to skip indirect block metadata when recovering files. Otherwise, the indirect block data gets mixed in with the recovered data, resulting in corruption in the recovered files.

Step 6c: Search for Strings

String searches useful for:

- Detecting "signatures" of known malware
- Finding "vocabulary" related to crime
- Finding specific intellectual property
- Locating deleted logs, mail messages, etc.

May be all that you're left with on modern Unix filesystems

Step 6c: Search for Strings

Sometimes string searching can be the best mechanism to zero in on certain data of interest to your investigation. For example, if you suspect a particular piece of malware, there may be embedded strings that you can look for to detect the presence of the malware on your system. Similarly, particular types of crime will often tend to use a specific "vocabulary"—e.g. bank accounts, victim and "money mule names", etc., for financial fraud—that you can search for. If you suspect theft of trade secrets, you could use string searches to look for stolen intellectual property in your seized images. And while tools like foremost may not be so good at pulling out "textual" kinds of data like log files and email messages, string searches are perfect for this.

Creating Strings Databases

- May need to do repeated searches
- More efficient to extract all strings first:

```
# mmls dev_sda.dd
DOS Partition Table
Units are in 512-byte sectors

     Slot    Start        End          Length       Description
00:  -----   0000000000   0000000000   0000000001   Primary Table (#0)
01:  -----   0000000001   0000000031   0000000031   Unallocated
02:  00:00   0000000032   0001884159   0001884128   Linux (0x83)
03:  00:01   0001884160   0002097151   0000212992   Linux Swap (0x82)

# dd if=dev_sda.dd bs=512 skip=32 count=1884128 |
            strings -a -t d >dev_sda1.asc
```


Creating Strings Databases

You will often end up running many different string searches as your case evolves. Having to constantly rerun your search over the entire disk image is wastefully time-consuming. Also, we need have our byte offsets be from the start of each partition, rather than from the start of the entire disk image so that our block number math works properly.

It is usually far more useful to extract all strings from each partition in your image first so that you have a smaller collection of data to search through and so you get more useful byte offsets. Here, we're using `dd` combined with the output of `mmls` to extract our Linux partition to the standard output. "`bs=512`" specifies that we're using units of 512-byte sectors to extract the data. Since our partition starts 32 sectors into the image, we "`skip`" the first 32 sectors before we start extracting data. The "`Length`" field in the `mmls` output tells us how many sectors long the partition is, and we feed that into the "`count`" option to `dd`.

The `strings` command will extract all strings of four or more ASCII characters terminated by a null (ASCII 0) or newline. "`-a`" ("all") means to search the entire input file, even though it will appear to be binary data, and "`-t d`" means to output the decimal byte offset of the start of each string (similar to the "`-b`" option with `grep`).

If you're dealing with a Windows image or a Unix image that has lots of Windows files in it (perhaps a file server, etc.), you will also want to use "`strings -a -t d -e l`" to extract the Unicode strings from your image. You could repeat the above `dd` command to do this, rereading the same data twice is wasteful—and time-consuming if you're dealing with a large image! The following article has instructions on how to use FIFOs to extract ASCII and Unicode strings at the same time:

http://computer-forensics.sans.org/blog/2010/01/27/fun-with-fifos-part-ii-output-splitting

Word List Searches (`grep -f`)

Have a "Dirty Word List" file with known signature strings:

```
# grep -if wordlist dev_sda1.asc >hits
```

Wordlist file format is one string per line:

```
rootkit
h4xx0r
gr33tz
psybnc
etc ...
```


Word List Searches (`grep -f`)

Once you've extracted the strings, you can just `grep` against that—no need for "`grep -b`" now since `strings` already output the byte offsets for you! It's likely that you will be looking for more than just one string, and grep let's you specify a file of patterns to look for with the –f option. Patterns should be unique words of six or more characters that are unlikely to result in false-positives.

The strings in your pattern file will vary depending on the type of case you're dealing with. For example, you might include different words if you think the attack on your system originated from China, South America, or Eastern Europe. The words you use in a child exploitation investigation are different from what you'd be looking for if you suspected an external attack and rootkit installation. Professional forensic investigators will have several different "dirty word lists" for different types of cases that they continuously update and refine.

Exercise 5: Unallocated Space

- Extract strings from disk image
- Create a **`grep`** pattern file and search for strings
- Use TSK and other specialized tools to recover deleted data

Exercise 5: Unallocated Space

Exercises are in HTML files in two places: On the course USB media, and in the home directory of the SANS user in the lab virtual machine. These two locations contain identical copies of the exercise files—it just depends on whether you prefer to look at them from a web browser inside the lab VM (because you're working in full-screen mode) or from your host operating system.

1. Open your web browser
2. Select "File … Open …" from the browser menu (Ctrl-O is the usual keyboard shortcut)
3. This is Day 6, Exercise 5, so navigate to *.../Exercises/Day_6/index.html* and open this HTML file
4. Click on the hyperlink, which is the title of Exercise 5
5. Follow the instructions in the exercise

Which Brings Us Full Circle to the First Exercise in the Course!

This page intentionally left blank.

Step 7: Create Incident Report

Incident report should contain both an executive summary and a detailed, technical write-up of the investigation

The report should include input *(and consensus)* from everyone on the incident response team

Keep in mind that it:

- Could be used in court
- Scrutinized by opposing counsel

Step 7: Create Incident Report

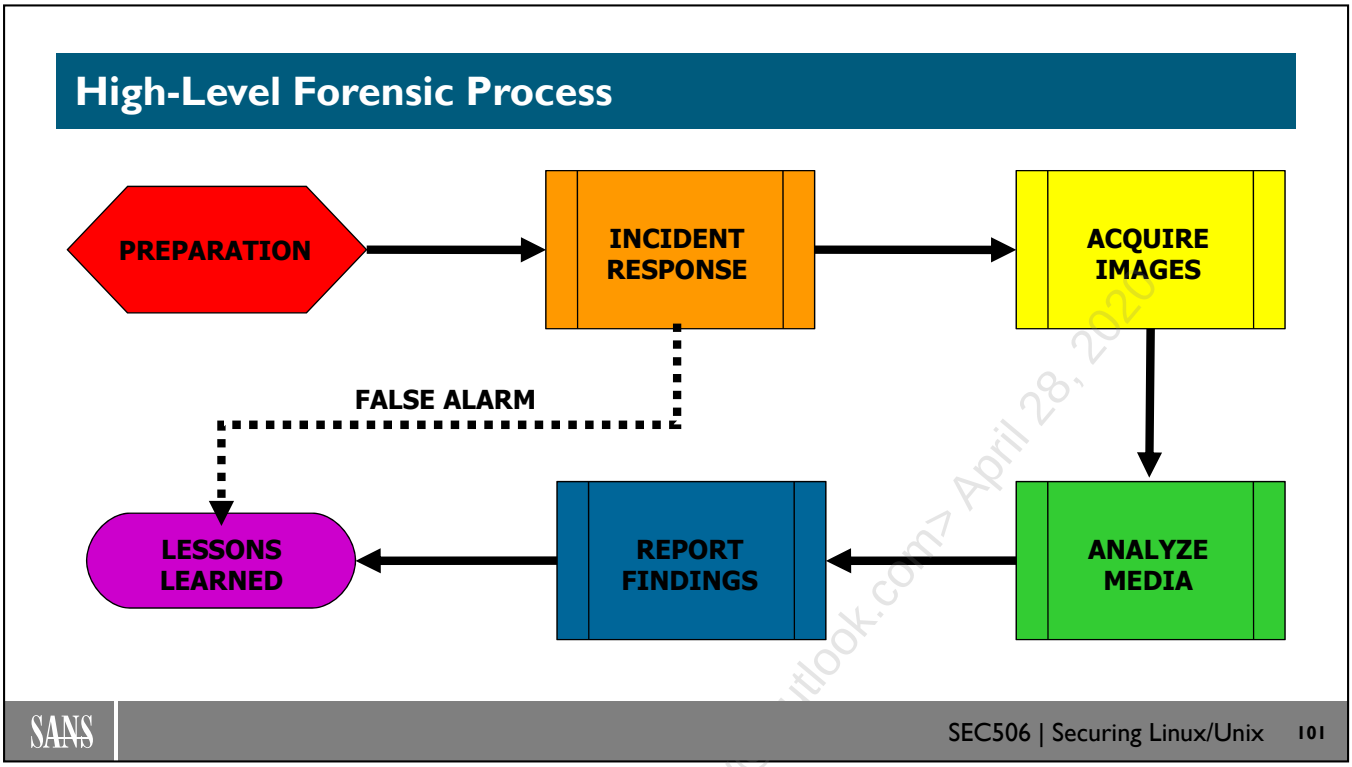
Reporting is not entirely a technical aspect of computer forensics, but it is probably the most important step of the forensic process. Most reporting is done to individuals who may not be educated in computer hardware, networking topics, or computer crime law. You need to ensure that your reporting clearly explains the evidence you found, the techniques you used and defines everything that is technical. Many investigators, unfortunately, overlook this easy step because they think that most people can understand how the internet works or even topics as simple as file downloading. However, in a court of law, or even in your own company, for anyone to utilize your results, they need to make sure that it is understandable. In a nutshell, document everything, explain and define basic and advanced topics, and show the results of your investigation.

Step 8: Apply Lessons Learned

- Every incident is an opportunity to get better
- We learn a lot about the overall incident response and forensic process with each investigation
- Failure to apply lessons learned can result in damage to professional or organizational reputation

Step 8: Apply Lessons Learned

Every incident is an opportunity to learn about security and its application within our organization. It may be that the lesson learned is that new policies are needed. It may be that a policy was ignored. It could also be the case that there was little the organization could do, in which case the lessons learned would revolve around better incident handling. Make absolutely sure that your organization learns from these lapses. Failure to apply these lessons could result in even worse damage to corporate reputation. After all, you can survive one web defacement, but how about if it happens again one week later? Maybe not.



High-Level Forensic Process

Again, here you see a simplified flowchart of the forensic investigation process. As we stated before, being adequately prepared to respond is absolutely critical, and if you spend enough time in this proactive phase, you can greatly reduce the stress that you encounter and the chance of making costly mistakes during the reactive process of incident response.

It is important to mention that not every incident is an intrusion or compromise, so we have incorporated a dotted-line to indicate that false alarms do occur. In these cases, the incident response team is activated to handle what appears to be an incident, but it later turns out to be something less serious—such as an improperly configured system or over-zealous user/administrator.

Image acquisition is the process of creating true bit images of the disks, tapes, and removable media that may be associated with the system in question. It is an art form unto itself, albeit an easy one to master. Once these images have been acquired, the original evidence (system disk) can be locked in a container for safe keeping. You will conduct your investigation against these images, performing media analysis. Examining the images for small pieces of evidence that will ultimately help you to reconstruct what happened in the past on a system.

The two final pieces are the ones that are most often overlooked. The creation of an accurate and effective incident report, and incorporating any lessons learned (especially after "false alarms") back into the security processes and policies of the organization.

Conclusions

- We have covered a tremendous amount of information about Linux Forensics
- There is MUCH more, but armed with this basic knowledge, you will succeed
- Forensics is 30% preparation, 70% perspiration
- Requires diligence, a keen eye, and persistence

This page intentionally left blank.

THANK YOU

Please remember to fill out your evals!

Hal Pomeranz
hal@deer-run.com
http://www.deer-run.com/~hal/
@hal_pomeranz on Twitter

Course Resources and Contact Information

AUTHOR CONTACT
Hal Pomeranz
hal@deer-run.com
http://www.deer-run.com/~hal/
Twitter @hal_pomeranz

SANS INSTITUTE
11200 Rockville Pike., Suite 200
N. Bethesda, MD 20852
301.654.SANS(7267)

CYBERDEFENSE RESOURCES
cyber-defense.sans.org
Twitter: @SANSDefense

SANS EMAIL
GENERAL INQUIRIES: info@sans.org
REGISTRATION: registration@sans.org
TUITION: tuition@sans.org
PRESS/PR: press@sans.org

Hal Pomeranz
hal@deer-run.com
http://www.deer-run.com/~hal/
@hal_pomeranz on Twitter

Index

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z