



SANS

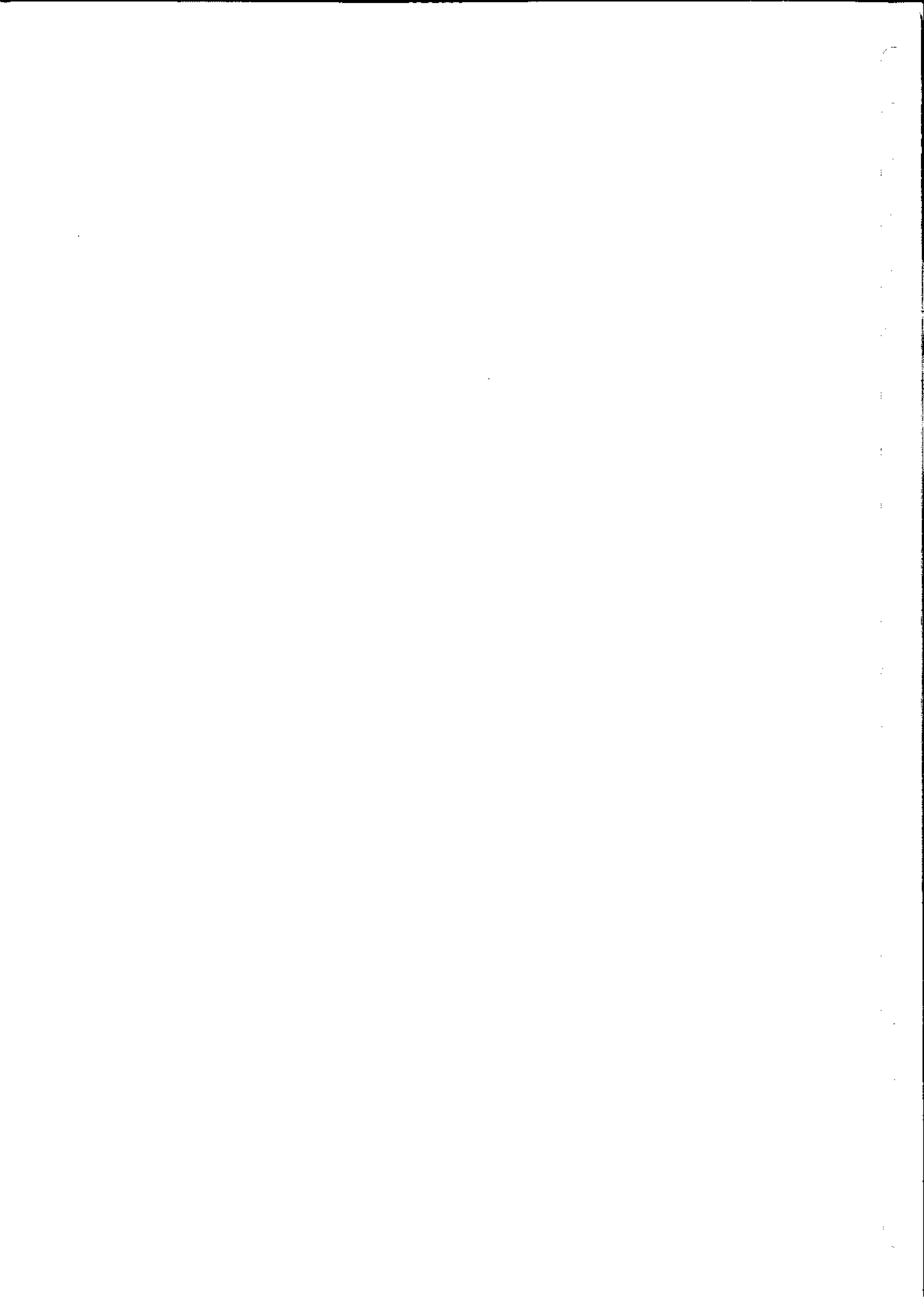
www.sans.org

SECURITY 503
INTRUSION DETECTION
IN-DEPTH

503.4

Open Source IDS:
Snort and Bro

The right security training for your staff, at the right time, in the right location.





SANS

www.sans.org

SECURITY 503
INTRUSION DETECTION
IN-DEPTH

503.4

Open-Source IDS:
Snort and Bro

The right security training for your staff, at the right time, in the right location.

Copyright © 2015, The SANS Institute. All rights reserved. The entire contents of this publication are the property of the SANS Institute.

IMPORTANT-READ CAREFULLY:


This Courseware License Agreement ("CLA") is a legal agreement between you (either an individual or a single entity; henceforth User) and the SANS Institute for the personal, non-transferable use of this courseware. User agrees that the CLA is the complete and exclusive statement of agreement between The SANS Institute and you and that this CLA supersedes any oral or written proposal, agreement or other communication relating to the subject matter of this CLA. If any provision of this CLA is declared unenforceable in any jurisdiction, then such provision shall be deemed to be severable from this CLA and shall not affect the remainder thereof. An amendment or addendum to this CLA may accompany this courseware. **BY ACCEPTING THIS COURSEWARE YOU AGREE TO BE BOUND BY THE TERMS OF THIS CLA. IF YOU DO NOT AGREE YOU MAY RETURN IT TO THE SANS INSTITUTE FOR A FULL REFUND, IF APPLICABLE.** The SANS Institute hereby grants User a non-exclusive license to use the material contained in this courseware subject to the terms of this agreement. User may not copy, reproduce, re-publish, distribute, display, modify or create derivative works based upon all or any portion of this publication in any medium whether printed, electronic or otherwise, for any purpose without the express written consent of the SANS Institute. Additionally, user may not sell, rent, lease, trade, or otherwise transfer the courseware in any way, shape, or form without the express written consent of the SANS Institute.

The SANS Institute reserves the right to terminate the above lease at any time. Upon termination of the lease, user is obligated to return all materials covered by the lease within a reasonable amount of time.

SANS acknowledges that any and all software and/or tools presented in this courseware are the sole property of their respective trademark/registered/copyright owners.

AirDrop, AirPort, AirPort Time Capsule, Apple, Apple Remote Desktop, Apple TV, App Nap, Back to My Mac, Boot Camp, Cocoa, FaceTime, FileVault, Finder, FireWire, FireWire logo, iCal, iChat, iLife, iMac, iMessage, iPad, iPad Air, iPad Mini, iPhone, iPhoto, iPod, iPod classic, iPod shuffle, iPod nano, iPod touch, iTunes, iTunes logo, iWork, Keychain, Keynote, Mac, Mac Logo, MacBook, MacBook Air, MacBook Pro, Macintosh, Mac OS, Mac Pro, Numbers, OS X, Pages, Passbook, Retina, Safari, Siri, Spaces, Spotlight, There's an app for that, Time Capsule, Time Machine, Touch ID, Xcode, Xserve, App Store, and iCloud are registered trademarks of Apple Inc.

Overall Course Roadmap

- 503.1: Fundamentals of Traffic Analysis: Part I
- 503.2: Fundamentals of Traffic Analysis: Part II
- 503.3: Application Protocols and Traffic Analysis
- 503.4: Open Source IDS: Snort and Bro 
- 503.5: Network Traffic Forensics and Monitoring
- 503.6: IDS Challenge

Intrusion Detection In-Depth

Now that we've covered the foundation of theory required to understand network traffic at the link, network, transport, and application layers, let's get to work on learning about some open source traffic inspection tools – Snort and Bro. Snort characterizes itself as an Intrusion Detection System (IDS) or Intrusion Prevention System (IPS). There is a tendency to want to call Bro an IDS. Yet, it is actually described by its authors as a tool or framework for network traffic analysis and inspection.

We'll approach both of these tools by examining their operational life cycle. This lifecycle brings you from planning to refinement of each of the tools, cycling through the iterative processes, such as updating.

Special recognition and thanks to Mike Poor and Marty Roesch for their valuable author contributions.

Open Source IDS: Snort and Bro

Intrusion Detection In-Depth

This page intentionally left blank.

Day 4 Roadmap

- Traffic Inspection Tools Operational Lifecycle
- We will cover the following phases of the lifecycle first in theory, then as they pertain to Snort, and finally as they pertain to Bro:
 - Planning
 - Installation
 - Configuration
 - Running
 - Customization
 - Auditing
 - Refining
 - Updating

Intrusion Detection In-Depth

Today, we are going to examine Snort and Bro in terms of their operational lifecycle. In other words, this means understanding what tasks are necessary, in general, to consider and perform when deploying these tools, or for that matter any other production software. Today starts with some theory about the phases of the operational lifecycle, including planning, installation, configuration, running, customization, auditing, refinement, and updating.

Next, we'll apply most of these phases as they relate to Snort and Bro. The more transactional phases such as installation, configuration, and updating for Snort and Bro are found in the Appendix of Snort Material and Appendix of Bro Material. The reason for this is because there is a lot of material to cover today and we want to present the topics that are more nuanced and more complex. The coverage of these phases will offer you a broad view of what is entailed for the production operation of each of these amazing open source tools. This is more than just a step-by-step discussion of install, configure, and run the tools. This approach provides a recipe for a successful deliberated deployment, not just a haphazard "download and install the code and hope for the best".

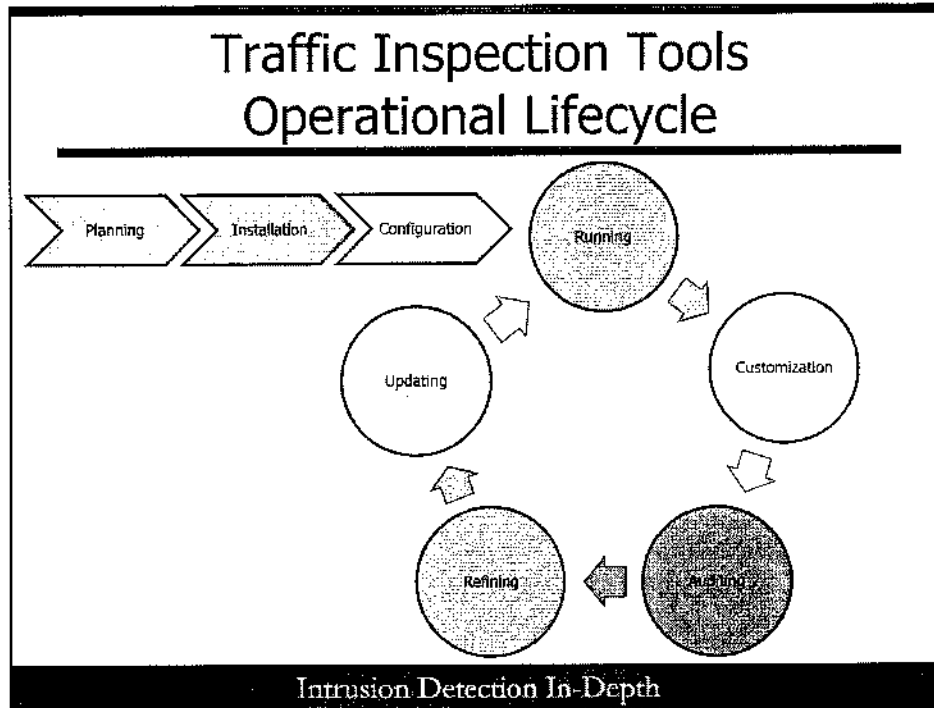
We include auditing in the theory section, but do not discuss the specifics of doing so with either Snort or Bro. Auditing is typically outside the purview of the analyst's duty, yet it is an integral part of the lifecycle.

We could devote a day or more to teaching about either Snort or Bro individually since they both have so many capabilities and features, some quite advanced. Unfortunately, we do not have the time to do so. That said, you should know that what you will learn from Day 4 material is a basic introduction to each. The intent is to give you the knowledge to intelligently run, configure, and customize each when you return to your office or home.

As discussed, there are individual appendices of additional Snort and Bro material that discuss some of the more straightforward phases. As well, some more advanced topics such as writing, testing, and implementing Bro scripts are in the Appendix of Bro Material.

We would like to cite the author or vendor of each tool in advance and give credit and gratitude to them for their contributions.

Snort	Marty Roesch, Sourcefire/Cisco
Bro	Vern Paxson, Seth Hall, Robin Sommer, Broala, research teams at International Computer Science Institute and National Center for Supercomputing Applications
Broala	Commercial Bro
ELSA	Martin Holste
PF RING	ntop team
Emerging Threats	Matt Jonkman
Barnyard2	Andrew Baker, Marty Roesch
Bro heartbleed.bro code	Bernhard Amann
Pulled Pork	JJ Cumings



Regardless what tool you deploy, whether or not it is for traffic inspection, there is an operational lifecycle process that guides what and when to perform specific tasks. As you are well aware not every manager or boss cares whether you follow the lifecycle process – most just want the IDS/IPS/whatever installed, running, and reporting in less than a day's time, and finding each and every hint of evil activity. Oh, that it were so easy!

We'll examine many of these different phases of the operation life cycle – first in theory, and re-examine most of the phases again using Snort, and finally using Bro. Some considerations are universal among any traffic inspection tool, such as finding and protecting the most sensitive or valuable assets on the network. Other phases, such as product installation and configuration are dependent on the specific solution you select.

There is an overabundance of technical detail associated with teaching Snort and Bro. Some of the topics such as installation, configuration and updating are straightforward; therefore they will be placed in the Appendices – one related to Snort, another related to Bro. You can use these as reference when you install, configure, or update the product. Topics that are considered less transactional in nature and require more theory are covered in the main sections of the course.

Planning

- One of the most often skipped steps
- Where are your sensor(s) to be placed?
- What are the network speeds at those places?
- What type of hardware do you need to handle those speeds?
- What capacity do you need now and the future?
- What best meets the site's needs – IDS or IPS?
- What type of software should you run?
- Do you need additional software/hardware for managing, backend processing, data retention, etc?
- Will IDS/IPS be run and monitored in-house or MSSP?

Intrusion Detection In-Depth

One of the most skipped steps in product installation is the first of planning the deployment. This involves knowing where you want the deployment(s) on the network(s), meaning that someone is aware of the "crown jewels". What needs to be protected the most? An answer of "everything" isn't exactly helpful. Does the site really need to examine all traffic? If so, you better be ready for a very broad configuration that encompasses a wide range and variety of traffic types. This is the generalist approach – see everything, but most likely at a high level. And/or are there parts of your site's network that need microscopic scrutiny, such as a part of the network that stores all customer data. So, the "where" must be answered predicated upon the "what" needs to be protected. A risk assessment of assets may be very helpful in determining sensor locations.

You must do capacity planning to determine your current needs in terms of number of sensors, management, and support software and hardware. Don't forget to consider your future needs while you are doing this. Bro makes future capacity requirements more easily extensible because of its flexible architecture, permitting the addition of new sensors with very little reconfiguration. However, other IDS/IPS solutions may not be as forgiving or flexible.

You will need to decide whether an IDS or IPS best meets the site's needs. It is possible to have a combination of both where the IPS is employed to guard assets of value that should have limited and tightly controlled access. It is less likely that a properly configured IPS will cause legitimate traffic to be blocked because of false positives in such an environment. However, a sensor(s) that protects a wide variety of disparate assets may best be configured as an IDS since the possibility of false positives may increase. Another option is to anticipate the use of an IPS, yet slowly phase in blocking as you initially configure it to log or alert. Once you become more familiar with the traffic and can eliminate many of the false positives you can begin to block.

Another planning concern is the hardware and software – perhaps considered individually or together. You may think that you are at liberty to select the hardware and software that you want to deploy, but layer 8 of the OSI model – politics - may intervene and stifle your best intentions. As well, even if you are allowed to select a product of choice and select some open source offering like Snort or Bro, your selection of hardware may be limited to whatever no one else wants. Obviously this is not ideal as you know and is hardly considered planning.

Some of the planning questions that need to be considered are – what rates of traffic flow must you examine, what type of hardware do you anticipate needing - including the NIC(s), bus, number of processors, what data is to be stored long term and for how long, and other processes associated with the traffic inspection that require other hardware, for instance a management or backend platform. This list is not all inclusive, but you can see good planning requires more than throwing a Snort box on the wire with its default configuration and rules.

You will need to consider what type of hardware to use to capture the traffic, such as a tap or switch. And, if you have a particularly high volume of traffic, it may be necessary to do some kind of load balancing to distribute the processing. We'll examine hardware concerns in more depth on Day 5.

A very important decision is whether to manage your IDS/IPS solutions in-house or use a Managed Security Service Provider (MSSP). The biggest advantage to managing the operation in-house is that you control everything about it; you are not dependent on hiring an MSSP whose competence and diligence are unknown except, perhaps, by reputation. The biggest disadvantage of managing the operation in-house is that you need to train and maintain competent and invested employees who have the incentive to stay. You will have to plan and budget for staff required to support the operation for the present and future two to five years away.

Installation

- Commercial solution may come preinstalled
- Install from source or bundle?
- Prerequisite software requirements?
- Build-time options
- Installation required on multiple hosts?

Intrusion Detection In-Depth

Once you've agreed upon the plans for the deployment the next step is to actually install the acquired product. Your participation in the process may be minimal or more extensive. Some commercial solutions may come preinstalled on some proprietary appliance and all you need to do is "plug it in" – put it on the proper place on the network, turn the hardware on, and start the software.

At the other end of the spectrum, especially in regard to open source software, your effort and involvement in the installation process is far more demanding. With open source software, you often have the choice of installing from source, namely configuring, compiling, and placing in system libraries. You may have the option of installing a precompiled bundle, rpm format, etc. While this is more convenient, you accept the configuration and build-time options that are set – perhaps not what you want. Consider whether or not you have experienced and talented analysts who can properly install, configure, and maintain an open source solution.

You not only have more control over the build and run-time options when you install the software from source, you also have more visibility into the code. Why would you even care about the ability of having the source code available for scrutiny? Suppose you select the bundling option for installation. You have no idea how the precompiled code was created or the code used to create it. This leaves more opportunity for some evil-doer to plant some malicious code. Sure, this is a paranoid attitude, but it warrants discussion and consideration.

As you are most likely aware, there are attackers acting on their own or in support of nation-states intent on infiltrating as many sites as possible. Certificate authorities, for example, have come under attack since compromise and access to the private keys for multiple certified sites allows impersonation of actual sites. This is far more efficient than trying to attack each site individually. So too is compromising a server that offers precompiled binaries for code that is used to detect attackers. The attacker need only compromise the site, add some new code to the existing code, compile it and offer up that malicious code on the server. Perhaps the malicious code adds a backdoor or tricks the user into believing that the software is running, yet generates bogus or no output. Imagine how dangerous that might be.

Compiling from source code is not a panacea since the attacker can simply put the malicious code into the existing source code. Most of us don't inspect the code to the extent of being able to find the attacker's code, yet there is more transparency if you choose to do so. You have the code at your disposal for examination if, for instance, you learn of a compromise to the site you used to download your code.

Another consideration when installing from source is the need for prerequisite software. Many open source projects have some required software that must be installed – for instance, at a minimum, libpcap or its equivalent to capture packets from the network. Sometimes the process of gathering and installing all the prerequisite software can be quite frustrating if you don't work from the base and methodically add, in order, the required software.

Another benefit of compiling from source is the capability to supply build-time options. Perhaps you may want to have the ability to debug a process that is not behaving as you expected. Perhaps an option to debug running code must be included at compile time. Otherwise, you have to accept whatever build-time options are included in the pre-compiled code.

Finally, you may find that you need more than a single traffic inspection solution. Even if you do not, a single deployed instance may require additional hardware/software for management and backend processing. In other words, you may have multiple hosts and products that need to be installed.

Configuration

- Take the default
- Do some rough rudimentary initial configuration
 - IP address/CIDR block of protected network
 - Define sniffing interface(s)
 - Add/remove processing for particular protocols
 - Initially, configuration should be overly inclusive
 - Remove rules for protocols/services not run
 - Initially, configuration should be overly inclusive
 - IPS should be configured to log not block initially
 - Initially, configuration should be overly exclusive of rules that block

Intrusion Detection In-Depth

Most operations require that you do a modicum of configuration before you run the tool for the first time. It may be possible to take the default setup, run it, and see what you get. Chances are there is some rough or rudimentary configuration that must be performed to refine from a generic one-size-fits-all to a site-specific design.

Some of the basic tasks of configuration may include defining the sniffing network interface(s) and assigning a configuration variable specifying the IP addresses or CIDR block of the protected network. It is best that the tool knows what is considered the protected network versus traffic from anywhere else.

Most products will afford you the opportunity to designate your running protocols or services to minimize the overhead of including processing for unsupported protocols. Initially, you may not be aware of all the site-supported protocols, but after running awhile and examining output, you may realize that your solution is performing unnecessary processing. At first, it is best to be overly inclusive of both processing and rules so that you do not get false negatives. It is a learning process that may be refined over time to eliminate unnecessary overhead.

Most traffic inspection products allow you to select sets of rules, signatures, scripts, or some kind of process that looks for malicious signs in the traffic. As with inclusion of processing of unnecessary protocols, inclusion of unnecessary rules may create some unwanted overhead and diminish efficiency. Again, you may not be aware of what rules should be included initially, but over time, you may have a better idea.

If you are running an IPS, make sure that your initial attempts of placing the hardware/software on the network are non-disruptive. In other words, you do not want to block much, if any, traffic to begin with. Often times, an IPS is placed into logging mode, making it an inline IDS, initially as a precaution so legitimate traffic is not blocked. As you become more familiar with your traffic and the way the IPS works, a more liberal set of blocking rules may be turned on. Eventually, you may turn more rules from logging to blocking behavior.

Most commercial IPS solutions contain far fewer rules that block than the same company's IDS solution rules that alert. The IDS solution typically offers a broad set of alerting rules, but may take those same rules for an IPS and alert on some and block on those that have very low rates of false positives. Obviously, the last thing you want to do is block legitimate traffic when the IPS is put in place. No doubt, there are skeptics or demanding users on your networks who are convinced that an IPS is an impediment only. You do not want to give them any reason to reinforce their errant belief.

A unique configuration can differentiate your site from one that takes the default settings. An attacker who relies on evasions or methods that depend on default settings has a better chance of being successful when the configuration is unaltered. For instance, there are default memory allocations or time-out values for Snort that can be abused to the attacker's advantage.

Running

- Take it for a test run
- Make sure you're seeing the traffic you want
- Make sure you're getting output
- When satisfied that proper procedure/commands in place, put in start-up routines

Intrusion Detection In-Depth

You are ready to take the tool for a test run at this point. The initial run may be an exercise in ensuring that you are seeing all the traffic you expect to see. If an option exists to log traffic/packets, or if you can concurrently run tcpdump, it may be beneficial to capture several minutes worth of traffic and examine the IP addresses and protocols to make sure you are capturing necessary traffic. Look for traffic to known and expected services such as DNS and HTTP at a minimum.

Next, look at some of the output, whether logs or alerts to make sure that the software is processing the traffic – not just capturing it. If you are so inclined, you would be wise to run some traffic that you know will cause an alert or log message to be generated. If you are comfortable enough with the product to write your own simple rule or script, send some traffic that causes it to do expected processing.

Once you are satisfied that the software is running well and is not problematic, you need to ensure that it will automatically start on each reboot.

Customization

- Configuration with a clue
- After initial installation and operation, becomes clearer how your site is different from generic configuration
 - Rules/signatures
 - Parameters or preprocessors/scripts
 - False positives
 - Backend processing
- Repeated whenever update or change in architecture, placement, etc.

Intrusion Detection In-Depth

You've run the software for some time and know it is processing the traffic and producing output – now what? Chances are when you did your initial configuration, you were thoughtfully guessing what you believed needed to be included and defined. Now that you have a clue (hopefully), you are ready to perform customization for your particular site and needs.

If you are able to examine the default or your configured set of rules or signatures, you may now have a better awareness that some are superfluous and can be excluded. For instance, there might be a set of rules for VoIP, but you don't run any VoIP in your network. Exclude those from the configuration. Also, if you are running an IPS, you may be prepared to change from logging to blocking status for those rules that you have determined reliably report malicious activity only.

You may have the option, such as in Snort, to supply additional customization parameters like designation of all your ports and servers associated with HTTP or other protocols. Snort includes preprocessors for many of the well-known services on the network. However, if you do not run those at your site, you can initially comment out those preprocessor statements in the configuration file. Bro has Dynamic Protocol Detectors that do a similar job as Snort preprocessing; you may not need to run all of them. As well, Bro has scripts that perform processing and detection. Some are default, others deemed "policy" that you can optionally include.

Think of false positives as job security! You know that they are inevitable, but only the savvy analyst knows how to determine what is causing them and customize to eliminate them. After seeing the same alert fire multiple times on traffic and investigating it – perhaps by comparing what the rule looks for with the packet(s) it fired on – you can determine what the issue is and change or comment out the offending rule.

You may now be ready to run some kind of backend software once you realize that ASCII or even binary logs are just not cutting it. Backend processing software that includes a database and GUI can assist you in visualizing the traffic or reporting it in a more digestible manner than simple log output.

And, remember that customization is not a one shot deal. Every time you do some kind of change – whether that is an update of the product, a change in the rules, a change in site architecture or sensor placement, customization must be performed again once you become familiar with the new environment.

Bro espouses a philosophy of being "policy neutral". This means that it decouples alerts with their significance to any given site. For example, perhaps you are alerted of some kind of DNS activity to a fantasy football website. For some sites, such as academic ones, this is not an issue. However, if there are strict network policies about using the Internet for non-work related activity, this may be a big deal. Bro does not come with "custom" definitions of the danger for a given event; it gives the user an opportunity to assign some kind of interpretation or significance to it. In essence, this offers you the chance for more extensive customization.

Auditing

- How do you know that your IDS/IPS is doing what you think it should?
- Audit measures the efficacy of enforcement
- Audit measures the gap between policy and enforcement
- An audit may be mandated by compliance regulations, although some kind of audit ought to be performed even if not mandated
- Auditor should not be familiar with IDS/IPS configuration
- Make sure you audit the hardware/software – not the analyst

Intrusion Detection In-Depth

The word auditing probably has you screaming "not my job!!!". It's a nasty, but necessary part of the lifecycle process. This may be a mandated process for compliance, performed by someone other than you or co-workers on the security team. But, like it or not, you have to be a participant because you are expected to be able to find alerts or signs of traffic that the auditors send.

How do you know that your IDS/IPS is doing what it is intended to do and is enforcing the environment as expected? And, no - this doesn't mean fulfilling some kind of compliance requirement. An audit measures how effective your enforcement is. It is what exposes the gap between the mandated policy and the actual enforcement of that policy.

Think of it this way. It seems that the postal service to my house is not so great. I'm not sure if it starts at the local post office branch that is too busy to answer calls or when they do, too surly to do anything about mail that has been delivered to "/dev/null". Regardless, our mail delivery (more like mis-delivery) person has difficulty matching the numbers and letters on the envelope with those on the customers' mailboxes. There have been far too many instances of a negative response to "did you get the letter/package/material I sent?". I'm tempted to start sending mail to myself – not because I feel lonely and neglected, but as a sporadic audit of the competency and effectiveness of the local postal service. In this case I play both auditor and inspector since in a sense, I send the traffic, and validate its return, exposing the gap between the two acts.

You may think that you've got your site perfectly configured for intrusion detection or prevention, but how do you really know? You may have performed some tests where the IDS/IPS responded as you expected. But, you've got the inside track – you know how it should respond.

This is where impartial auditing comes in. The auditor or person who sends the malicious or noteworthy traffic should not have any knowledge of your configuration other than to know a destination IP address on the protected network. Two types of auditing are possible – one where the security team is forewarned of the audit,

or a blind one where the team is totally unaware that they and their IDS/IPS are under scrutiny. Personally, I believe it better to audit the IDS/IPS and not the analyst so the foreknowledge that an audit will occur at an approximate time will allow the analyst to be circumspect about the alerts and put the focus on the IDS/IPS where it belongs and not whether or not he's awake and alert at the time of a surprise audit.

SANS instructor Mike Poor and his company InGuardians are called upon to perform these types of audits. Their method is to inform the analyst of the exact time and nature of the malicious traffic they are going to send. They actually inform the analysts of the scenario of their attack. This may seem counterintuitive, but their intent is to let the analysts know what will occur and have the analyst(s) tell the story of the attack via the observed detection. InGuardians philosophy is that there is little chance an analyst will be able to understand and be able to recreate the traffic or methods of an unknown attack if the analyst cannot tell the story of a known attack.

Refinement

- Fine-tuned customization
- Speed improvements
 - Performance of hardware
 - Amount of RAM
 - NIC speed
 - HDD speed
 - Bus bandwidth connection of the above
 - Improved packet capture speed using PF_RING
- Efficiencies in rules/scripts
- Preventing false positives/negatives

Intrusion Detection In-Depth

You are almost finished once you've performed all the previous steps – traffic capture, and site-specific customization, etc. But it takes practice and experience to understand issues native to your site – those more complex than customization. Both Snort and Bro have built-in routines that can be run to measure efficiency and packet drops.

You may discover that processing speed becomes an issue and you are dropping packets. Resolving this issue may require a comprehensive inspection of all the component parts of the operation – hardware performance in terms of CPU speed and number of processors, the amount of RAM, NIC speed capability, Hard Disk Drive (HDD) speed and performance for rapid data storage, and the bus bandwidth that facilitates communications among these components.

Another performance improvement may be gained by installing software called PF_RING that accelerates packet capture, especially where bandwidth rates are greater, by using a data structure called a ring buffer along with shared memory. PF_RING also reduces the load on the kernel for every captured packet that results in improved packet capture efficiency and performance. As we will see both Snort and Bro can use PF_RING. Snort is able to use PF_RING to emulate a multi-threaded environment by balancing flows, thereby realizing performance improvements. PF_RING allows Bro to do host load balancing of traffic received from multiple sniffing interfaces.

More information about PF_RING and how to install it can be found at:
<http://www.metaflows.com/technology/pf-ring>

In general, you must be aware of the types of processing performed, rules included, and inefficiencies in rules or scripts, especially those that are home grown – either by you or your team or perhaps at some open source rule distribution site. This whole process requires someone knowledgeable not only about the configuration/customization of the product, but the type of traffic observed, as well as past customizations.

Refinement is a fine-tuned iterative process of taking a site-specific customization and tuning it for optimal performance. It also includes observing the output over an extended time and honing the rules and scripts to work in your unique environment. This is different than the more coarse customization of selecting appropriate rules and scripts. This requires that you tweak the imperfections out for your configuration not only to gain efficiency, but accuracy as well.

Updating

- New releases
- New rules/scripts
- New management software
- New backend software
- Perform related previous steps again
 - Customize
 - Refine

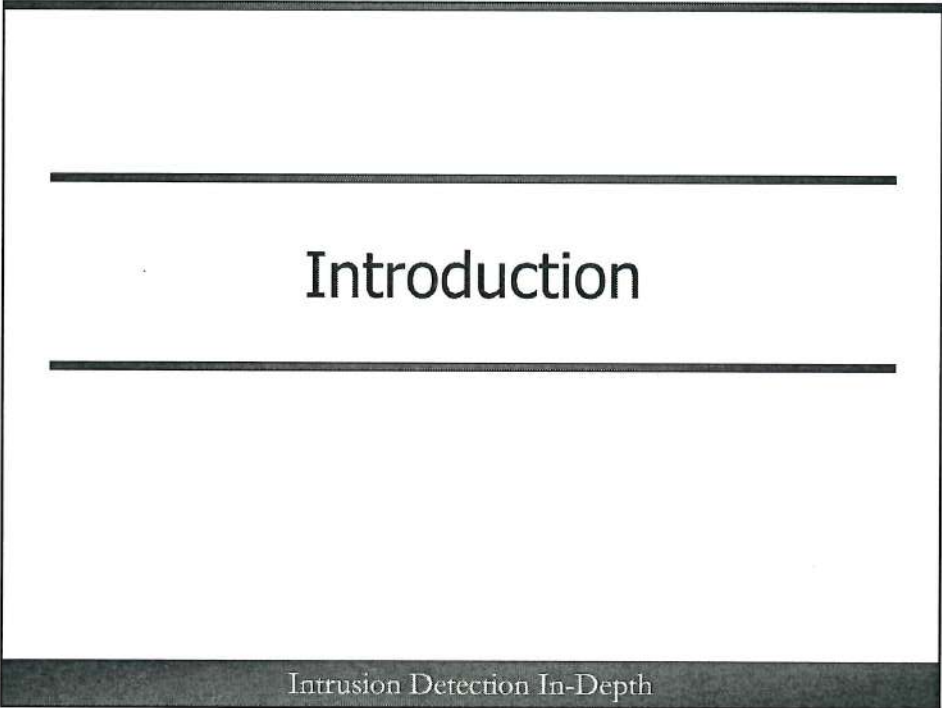
Intrusion Detection In-Depth

Okay – we're almost done; everything is running efficiently and you are the resident expert and hero(ine) for protecting your network from evil. And then, an updated software release is announced. Perhaps you don't jump on the upgrade right away, but eventually, either due to end of life expiration dates, inability to use new features, or just good maintenance practice, you have to update your software. Once done, you'll have to customize and refine again for the new update.

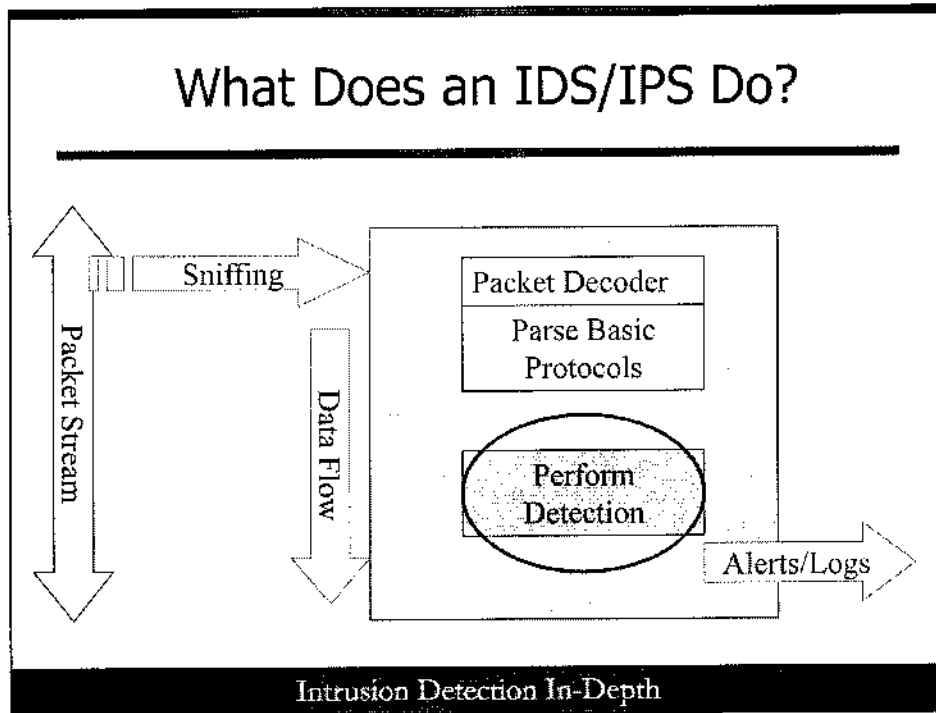
You'll find rule updates are released more frequently than new software upgrades. The two may actually be synchronized in some way where you need to upgrade software in order to use the new features or rules. Regardless, you need to stay current with new threats. Most products have automated ways to update rules so the burden is not the update process itself; it is the effect of updating the rules. You should be more mindful of performance and false positives each time new rules are added.

You'll have to be aware of updates of any kind of deployed support software, perhaps that allows you to manage the product or examine the output – for example a management console/GUI. There may be some adjustment to support software updates to ensure that you are generating the same type of reports and able to do the same type of manipulation and navigation of output to examine traffic as you've become accustomed.

Remember that the updating process causes changes – some predictable some not. Therefore, you may have to tweak current customizations and perform some refinement after observing the effects of the upgrade.



This page intentionally left blank.



Before we get into how Snort and Bro work, it is helpful to look at an overview of the job of the IDS/IPS and how it works at a rudimentary level. It must first sniff the traffic from the network. Once the packet is acquired, it must be analyzed into its component layers, such as IP, transport, etc. And, there is usually some additional processing performed that prepares the dissected packets for the detection phase. Common processing includes reassembling any fragments, reassembling TCP streams, and parsing some protocols – most likely those that are often used such as HTTP, and DNS to name a few. Really, there is a lot of preprocessing under the hood before detection begins, and for our discussion on this slide we'll assume that part is working, even though some solutions do it better than others.

These phases that occur before and after detection are important and most function well for the IDS/IPS to do its job, but most of the focus for us as analysts is the detection part. There is an output phase that follows the detection where detections are somehow processed for viewing or further analysis. Not to diminish any of the other phases, but the meat for us is the proficiency of detection and our ability to interface with it.

The Analyst's Role in Detection

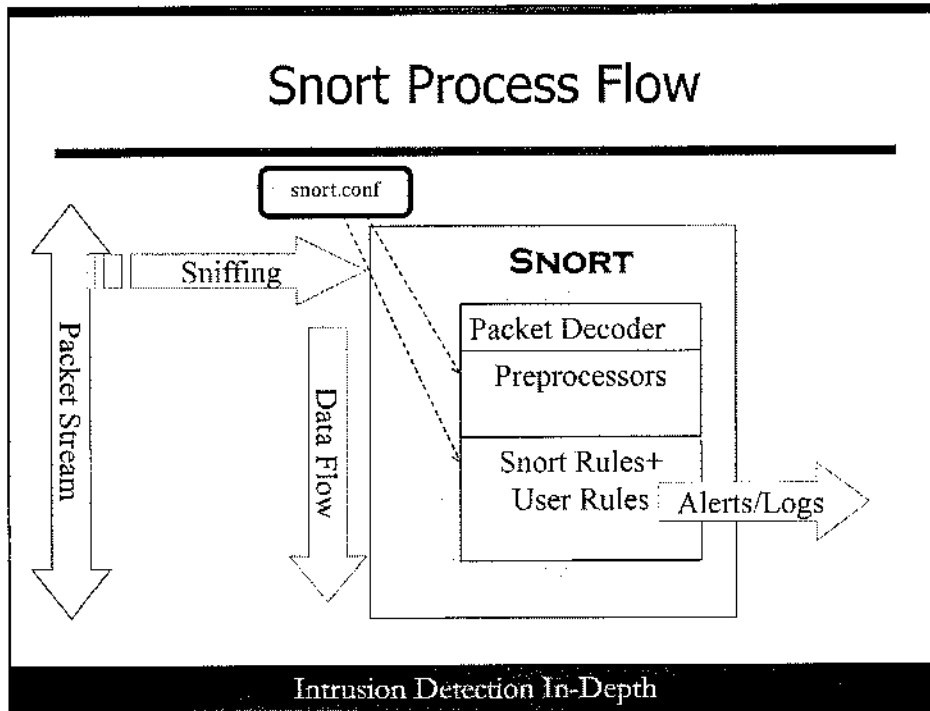
- Best detection offers the analyst
 - Visibility into the detection process
 - The capability and "language" that permit you to manipulate detection
 - Snort – rules
 - Bro - scripting

Intrusion Detection In-Depth

Your role in detection is very important because it is the most critical phase for an IDS/IPS and ideally you have some control over it. You are a key component in the success of the IDS/IPS operation. This means that you need visibility into the detection process. If you can't see or understand what the detection is doing, you have no idea whether or not you are covered for a particular threat. And, you will not be able to determine which alerts are false positives and eliminate or at least suppress whatever is generating them.

We need the capability to manipulate or control the detection process. This is provided by some "language" that allows you access to the detection process. The "language" is a very important part. In essence, we need a part of the detection process "exposed" to us through some means.

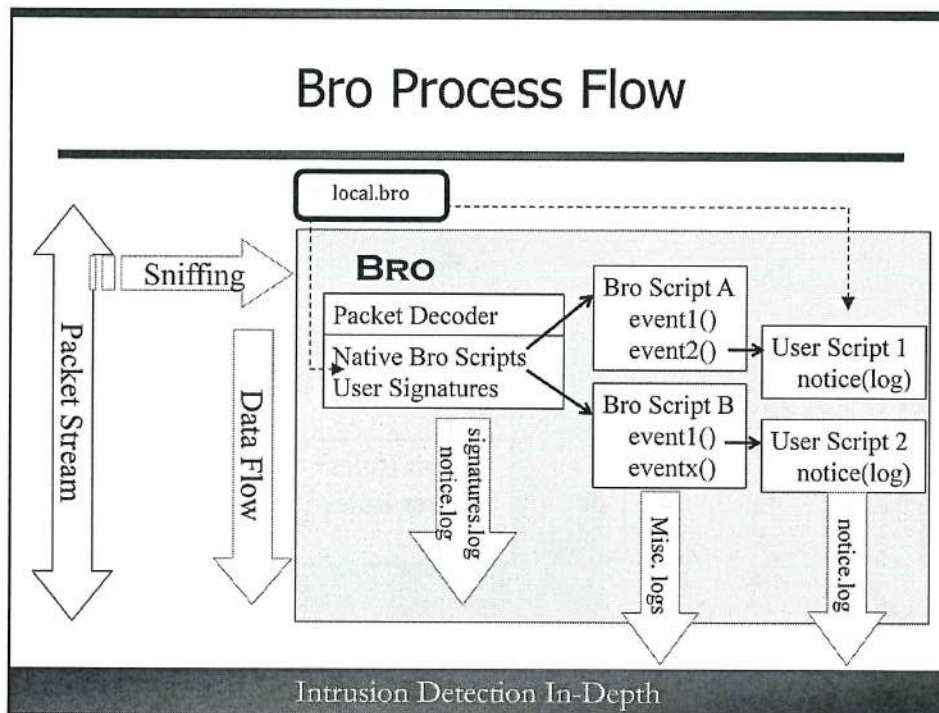
In our discussion of Snort and Bro, we will discover that the fundamental "language" that Snort uses is rules, whereas Bro uses scripts in a language written expressly for it. Snort rules provide us a "language" allowing us to designate packet characteristics of interest. In essence, Snort exposes a more user-friendly "language" and does most of the processing of the rules under the hood. On the other hand, Bro exposes the entire "language" to us. While this permits far more manipulation of the detection process, it is a harder "language" to learn since you must learn the language itself and figure out how it can accomplish what you want.



This slide shows how Snort processes traffic. The traffic is sniffed from the network interface and passed to Snort. The packet is decoded and then different preprocessors prepare the traffic to be examined by the rules by reassembling streams, and parsing the embedded protocols, for instance. Then rules that come with Snort and any user-supplied rules examine the traffic for noteworthy activity. The packets and alerts associated with this activity are placed in alert and log files.

A configuration file named "snort.conf" is included in the Snort install. It tells Snort which preprocessors to enable and which rules to include. There may be certain preprocessors that a particular site needs along with specific rule sets that are pertinent for the environment. The user tunes "snort.conf" for the site environment.

This is a skeletal view of the process that we will examine in more detail to cover each of these phases and processes.



This slide shows the way Bro processes traffic. Bro decodes the packet and invokes some native scripts that analyze and process the packet protocols much the way Snort preprocessors do. While simple signatures can be written by the user, Bro includes none. The native Bro scripts consist of code and logic and they define events. Events are a very important part of Bro.

Events are "mostly high level network events that protocol analyzers detect" according to the file names where they are stored. Don't think of events as alerts or even as something that should not occur. In fact, these are parts and states of protocols. For instance, there is an event for a DNS query, or one for an HTTP request. Most encountered events log the packet associated with it – such as the DNS request. However, a user can write a script to do something after a particular event is triggered. This is optional code. A user script can only be triggered when a user-selected event occurs. Think of an event as the means that Bro uses to allow further processing by the user.

Typically a user script contains some kind of logic examining a particular facet associated with the event. As an example, it might look for a DNS query for address resolution of "www.evil.com" after you learn that is a known malicious site. Bro processes the packet and follows the code path (the logical path of execution followed that is based on the code and the characteristics of the data it is processing), to analyze a DNS packet. On the way, it encounters Bro-defined milestone events. It may have encountered some logic to discover if this packet generated a "new_connection" event, but it was not triggered since this is not a new connection. Had it been a new connection, this particular event would have created an entry in a connection log. If there was user-defined code that triggered from the event, the code path would be directed to it.

Bro eventually encounters the "dns_query" event as it continues to process the DNS request packet. The "dns_query" event causes Bro to log the DNS packet data in its DNS log. As well, Bro knows of the existence of our script that is triggered by the "dns_query" event. The code path invokes our script that was loaded upon Bro invocation. Our script references the "dns_query" event and contains the post-event processing of matching the query seen in the traffic with "www.evil.com". Upon matching, the user can perform some kind of action,

defined as raising a notice, say to log the traffic. This allows the user to customize the action and log message - assigning it a notion of risk, importance, danger, etc. specific to the site.

As Bro is processing a packet/stream, it is also matching characteristics with those found in its simple signatures. A signature that fires creates a log entry in the "signatures.log" file.

Logged data from events is recorded in miscellaneous files depending on the activity. This could be new connection event data, something specific to the protocol in the traffic, as well as other miscellaneous Bro-defined events, or user triggered event processing that records entries in logs. Finally, the notices that are raised by the user scripts are entered in "notice.log". Bro native scripts can record entries in the "notice.log"; however these tend to be more informational, not necessarily indicative of important activity.

All the scripts and signatures that are loaded upon startup are found in a file called "local.bro".

Preview: What Snort and Bro Have in Common

Open source
Intended to be run on commodity hardware
User-customizable
Means of adding your own code
Produce ASCII output – alerts/logs
Rule/signature support

Intrusion Detection In-Depth

It is helpful to have some idea of the similarities and differences of Snort and Bro before we begin. This gives you some idea of the highlights of each product to be covered. You may not fully understand some of the concepts until we cover the material more thoroughly.

Both Snort and Bro are open source and both are intended to be run on commodity hardware. Both have many ways to customize the product for your site's particular needs. And, they both have a means for you to add your own code. Snort provides ways to hook into existing code so that you can write your own code for additional processing. Bro has its own scripting language that allows you to customize code from a triggered event.

Both produce voluminous amounts of ASCII output in the form of Snort alerts or Bro logs. Both benefit from having some kind of third party backend software to display the output in a more readable fashion, ideally with a means of correlation and graphing capability.

Finally, both have support for what they call rules or signatures. Snort has a very complex rule language, based mostly on finding content, yet with many more features. Bro has a simple rule language with basic content search capabilities. Bro relies more on its scripting language for creation of code to find events that require complex logic.

Preview: How Snort and Bro Differ

Snort	Bro
GNU GPL v.2 license	OpenBSD license
IPS implementation support	No true IPS implementation
Signature-driven	Event-driven
No native support for easy physical expansion of IDS/IPS coverage	Notion of Bro cluster makes physical expansion of IDS coverage simpler
Primarily provides intrusion/anomalous activity analysis	Provides a framework for network analysis
Support for creating sophisticated rules	Support for basic signatures, support for more sophisticated detection via scripting
When a rule fires, priority assigned to activity, one-size-fits all policy	Policy neutral, triggers on events with no assigned notion of good or bad

Intrusion Detection In-Depth

Snort comes with a GNU General Public License version 2 that places some limitations on what you can do with the code. Bro operates under the much more liberal OpenBSD license, permitting the code to be used in any way. Snort has support for inline IPS implementation, though Bro does not.

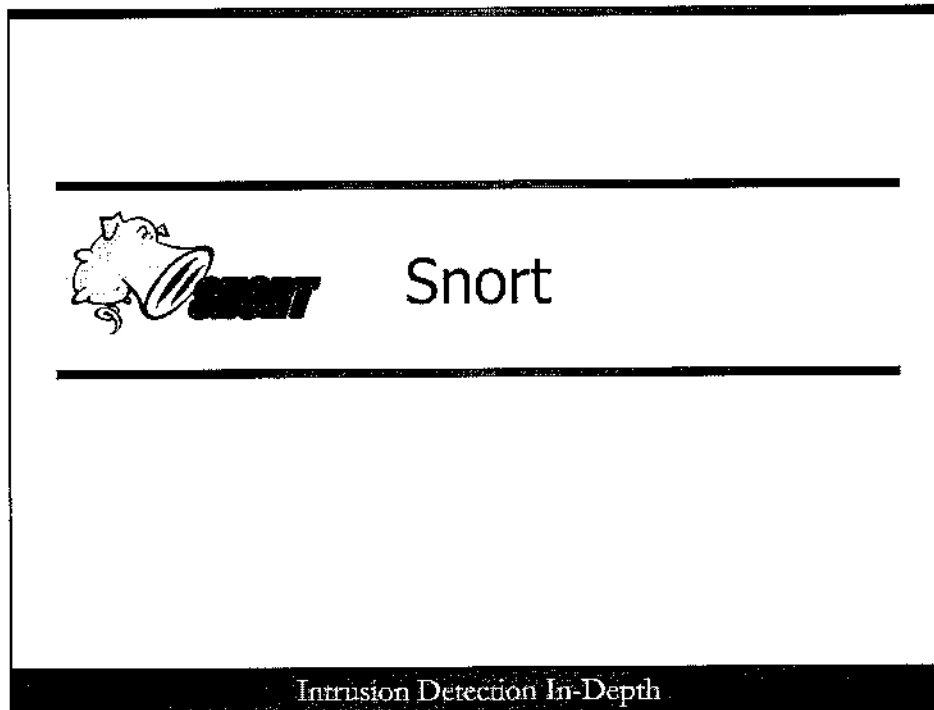
Snort triggers mostly based on signatures while Bro is event-driven. What this means is that Snort alerts generated from rules usually denote that there is something anomalous, noteworthy, or malicious happening. Bro's event-driven model defines events that can be recorded and acted upon. These do not necessarily imply that something malicious has occurred. For instance, Bro has an event for every new session connection to record in its connection log. The user can customize code for post-event activity and what, if any, action should follow.

Snort does not natively provide a means of constructing a network of Snort boxes with communications among all of the Snort hosts. If a site requires more coverage, it must add another implementation of Snort in the appropriate spot on the network. Bro has a notion of a cluster that involves the concept of many sensors/workers deployed and under the control of a management host. If additional sensor coverage is required, a new sensor/worker is added either on the same or new hardware. A simple configuration change is made to the manager to define the existence of the new host, making it an easy extension to the current Bro network.

The main purpose of Snort is to discover intrusions or anomalous activity directed to a host(s). Bro is able to do this as well, yet it records events and network connections to augment the visibility of the network as a whole, thereby enabling network traffic analysis. Herein lies the term "traffic analysis framework" – Bro is not just an IDS.

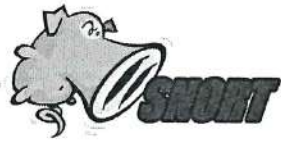
As mentioned in the previous slide, both Snort and Bro have rule/signature support. Both are capable of finding payload content. Snort has a sophisticated rule language, where Bro's is basic. Yet Bro offers the user a powerful scripting language that can do complex processing.

Finally, Bro touts itself as being "policy neutral". This means that the detection of some activity is decoupled from the interpretation of the importance or danger of the activity. It is up to the user to not only assign some judgment about the nature of the traffic, but also to define what to do upon detection. Snort, on the other hand assigns a message and priority value that interprets the triggered activity in a universal manner.



Snort has been a favorite, and widely used, intrusion detection/prevention system. Its primary focus is to inform you of anomalous conditions, or ones for which there are Snort signatures, or Snort preprocessor code for different protocols and protocol violations.

While Bro has actually been around longer than Snort, Snort has been more universally deployed and welcomed into networks everywhere. One of the reasons for this is because it is more user-friendly than Bro. Both can have a steep learning curve depending on your knowledge of protocols and how advanced you want the detection to be. But, generally Snort is easier to learn for the average user.



Planning

Intrusion Detection In-Depth

This page intentionally left blank.

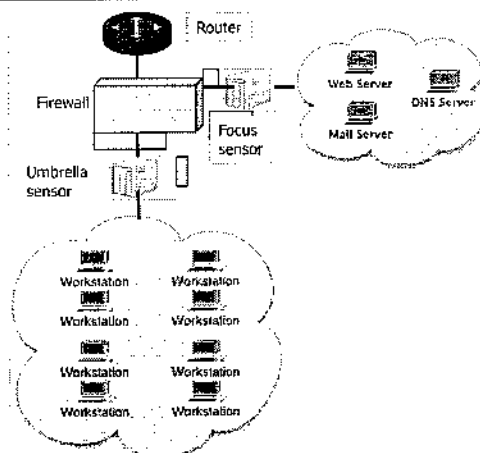
Snort/IDS Deployment Scenarios

Focus IDS Deployment

Focused rule set
Focused variables
Focused alert potential

Umbrella IDS Deployment

Wide rule coverage
Wide variables
Heavy alerting



Intrusion Detection In-Depth

When deploying any IDS/IPS, consider the two philosophies above. First, when deploying a focused sensor on a controlled network link, you can tailor your deployment to cover the most critical assets. There is a focus IDS at the top that is deployed to protect the site's DNS, mail, and web servers.

The rules on a focus sensor should be targeted to the traffic you expect to see on your network. For instance, if there are no FTP servers, you can comment out any rules or preprocessors associated with FTP. It is wisest to comment out instead of deleting the configuration lines that include FTP rules and preprocessors, after all someday you may deploy FTP servers. If you use any kind of software to track changes, document your alterations so that if you do deploy FTP servers, you or someone else will be aware that the current configuration does not cover FTP. As a matter of policy, tracking all changes is most beneficial, especially when the only employee who knows anything about Snort leaves the company.

The configuration variables specify those related to the IP addresses and characteristics associated with the focus network. If the configuration and rules you select are tailored properly, you should have less chance of false positives, especially on a single-purpose protected network.

Conversely, when deploying a sensor monitoring your primary operational network, in other words a link that monitors traffic where you can expect everything and anything, a wider approach is called for. On this "umbrella" IDS, expect heavy alerting with a higher degree of false positives. It is more difficult to be precise on such a varied network. Rules and preprocessor selection are general and broad in scope since the intent is to cover a wide and heterogeneous range of activity. Typically, the default Snort configuration with minor modifications is used on an umbrella sensor.

These deployment philosophies can be used for any monitoring IDS; they not unique to Snort.

Focus Sensor Configuration: Variables, Preprocessors and Rules

Variables

```
ipvar HOME_NET 10.10.10.0/24
ipvar EXTERNAL_NET !$HOME_NET
ipvar DNS_SERVERS 10.10.10.53
ipvar SMTP_SERVERS 10.10.10.25
ipvar HTTP_SERVERS 10.10.10.80
```

Preprocessors

```
preprocessor http_inspect_server: server default \
  profile all ports { 80 8080 } oversize_dir_length 500
preprocessor frag3_global: max_frags 65536
preprocessor frag3_engine: policy linux
preprocessor stream5_global: track_tcp yes, track_udp yes
preprocessor stream5_tcp: policy linux
```

Rules

```
include $RULE_PATH/local.rules
include $RULE_PATH/bad-traffic.rules
include $RULE_PATH/exploit.rules
```

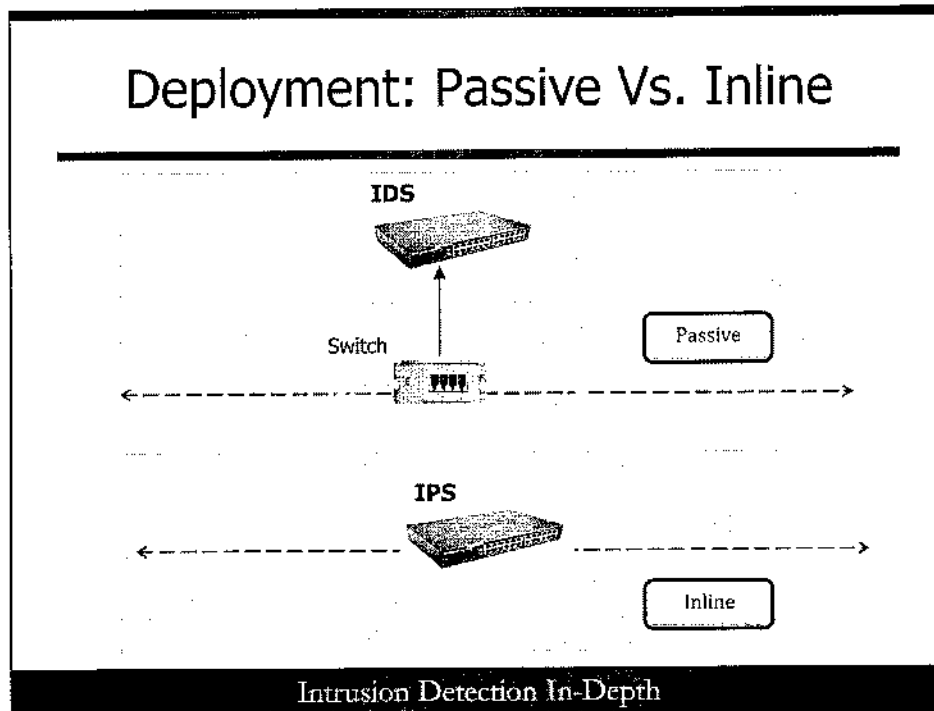
Intrusion Detection In-Depth

We will cover the Snort configuration file later for designation of variables, preprocessors, and rules. We introduce it here just to make you aware that these are the types of alterations that you will need to address when configuring the focus IDS/IPS. Specific IP addresses and ports should be designated for variables and preprocessors. When these variables are referenced in rules, Snort knows exact IP addresses and ports to examine. As you can imagine, this adds efficiency to the processing since Snort examines packets with these IP addresses and ports only, ignoring all others.

Here is a default set of preprocessors with changes to `http_inspect` for HTTP decoding to include traffic on ports 80 and 8080 since that is what our focused network runs. As well we've set both the `frag3_engine` for fragmentation reassembly policy and `stream5_tcp` stream reassembly policy for Linux because it is the only operating system found on our focused network. Other preprocessors such as the `ftp_telnet` would be commented out since we don't expect FTP or telnet traffic on this particular network.

As far as rules are concerned, we have our own custom rules in "local.rules", we use Snort's "bad-traffic.rules", and "exploit.rules". This is just a subset of the rules included in the configuration file since we would be wise to include any rules associated with DNS, SMTP, and HTTP for the servers on our focused network.

Deployment: Passive Vs. Inline



Snort can be deployed as a passive sniffing sensor that alerts and logs noteworthy traffic. This is the default mode. Another option is inline where traffic actually traverses two interfaces of the Snort box. This method allows traffic to be blocked, optionally.

You or your Computer Security Officer (CSO), if you have one, will have to determine the mode of operation. Passive mode is less likely to perturb the network flow of traffic, but the downside is that you learn after-the-fact that a noteworthy event has occurred. Inline provides better protection, but at a cost of potentially blocking good traffic for rules that are poorly or not precisely written. Always use caution when adding new rules to the inline configuration where prudence recommends placing the rules in logging mode before actually determining that they are candidates for blocking.

If you have several different sensors on your network, you may consider a mix of passive and inline where the passive ones are deployed in an umbrella mode to warn of potentially harmful traffic, but not perturb it in any way. As you deploy focus sensors for the assets on your network that typically cover the more sensitive functions and data, consider placing those in inline mode. The focused coverage allows you to be stingy about the rules you employ in the first place, and of those, the ones that you consider to be stable enough to actually block traffic.

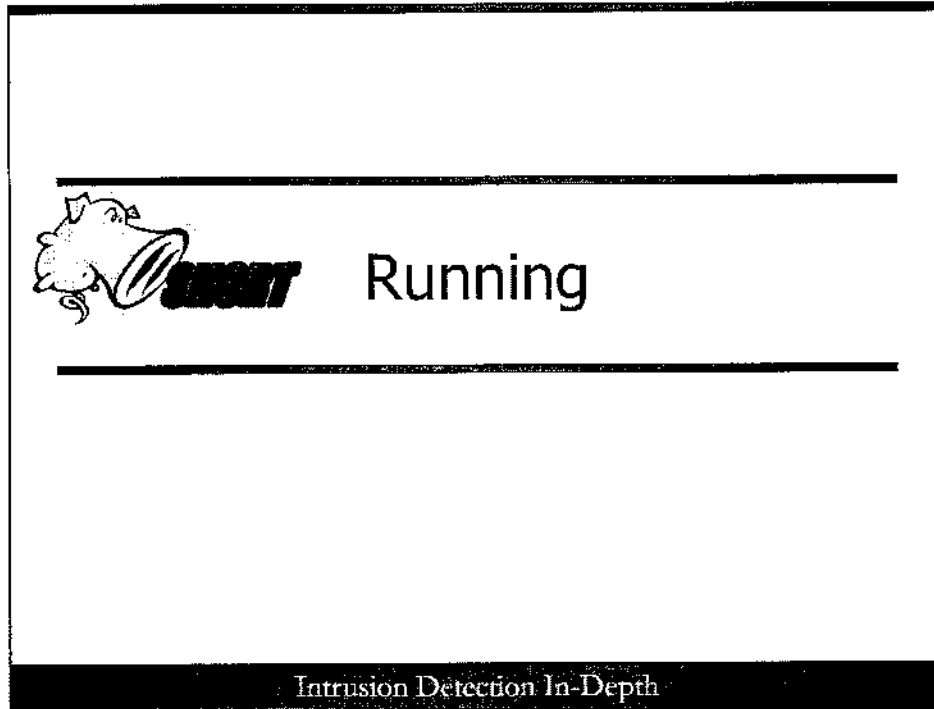


Installation/ Configuration

See Appendix of Snort Material for
Installation/Configuration material

Intrusion Detection In-Depth

This page intentionally left blank.



This page intentionally left blank.

Modes of Operation

- Snort has three general operational modes
 - Sniffer
 - Packet logger
 - NIDS
- Runtime mode is determined by command line switches

Intrusion Detection In-Depth

There are three basic modes of operation in Snort - sniffer, packet logger, and NIDS (Network Intrusion Detection System). Each one of these modes is well suited for a particular traffic analysis task. When running Snort, the operational mode is determined by the set of command line switches.

Sniffer mode is rarely used since it is akin to tcpdump in that it sniffs packets and prints them to the screen. It might be used for quick debugging purposes to make sure you are seeing the traffic you believe you are capturing.

Sniffer Mode

- Sniff and dump packets to screen
- Runtime command line switches:

-d: dump packet payloads

-X: dumps entire packet in hex

-e: display link layer data

- Example:

```
snort -de
```

Intrusion Detection In-Depth

Sniffer mode reads every packet off the network and dumps it in a decoded human-readable form to stdout (usually the screen on most systems). To enable sniffer mode, use the -d command line switch. This will cause Snort to display the packet, including headers and payload, to the screen.

There are two other options which are used less frequently. The -X option displays the packet in hex; and the -e option displays the link-layer. Combining all the switches together will give a very detailed, and overwhelming, display of the network traffic.

This is mostly useful in controlled environments or when you are testing to see if you are receiving the traffic you expected. Such an occasion might be after installation of new network sniffing hardware like a tap or switch. In a production network it's not going to be particularly useful unless you filter what Snort is looking for using Berkeley Packet Filters on the command line. They can be used in the same manner as tcpdump to filter specific traffic.

Snort Packet Dumps

What does all that stuff mean?

Timestamp 04/20-18:00:41.082504		IP Address:Port 192.168.1.5:4340 -> 192.168.1.3:23				
Protocol TCP	Time To Live TTL:64	Type Of Service TOS:0x0	IP ID ID:9409	Header Length IpLen:20	Datagram Length DgmLen:154	Frag Bits DF
TCP Flags *****S	TCP Sequence Seq: 0x6FF363B	TCP Ack Ack: 0x0	TCP Window Size Win: 7D78	TCP Header Length TcpLen: 32		
TCP Options TCP Options (3) => NOP NOP TS: 1860909 193728						

Intrusion Detection In-Depth

The displayed packets are broken out by layer into ASCII representation of the data contained within. The first line of the packet dump contains the critical data for determining when the packet arrived, where it came from and where it was going. The timestamp represents the date and time that the packet was seen (according to the system clock on the sensing host) down to the millisecond. The remainder of the first line, through the last field of the second line, are many of the other fields and values found in the IP header. The final line contains fields and values for the transport layer – TCP in this example.

Packet Logger Mode

- Tell Snort to output packets to a log file
- Command line options:
 - K ascii -l <logdir>: Dump packets in ASCII into <logdir>
 - l: Log packets in tcpdump binary format (default format)

- Examples:

```
snort -l /usr/var/log #Logs in binary
snort -K ascii -l /var/log/snort #Logs in ASCII
```

- Binary logs can be read back through Snort:

```
snort -dr /var/log/snort/snort.log
```

Intrusion Detection In-Depth

In packet logger mode, Snort records all packets that it sees to a file or set of files and directories. The packet logger mode is enabled with the `-l` switch on the command line. This switch will log packets in libpcap binary format which is the default mode.

When started with the combination of the `"-K ascii"` in conjunction with the `-l` switch, Snort starts logging packets into individual subdirectories named for the source IP address in the packet. Then a subdirectory is created under each of those directories named by protocol and port numbers for TCP and UDP or a name describing an ICMP packet such as `"ICMP_ECHO_REQUEST"`. All packets with matching characteristics are logged in that file.

For instance say there is a source IP of `192.168.11.42`. Any other packet that shares that same source IP is logged in a subdirectory. Suppose there is a TCP session with a source port of `53257` and a destination port of `80`. If you look in the directory designated for logging, you will find a subdirectory with the name of `"192.168.11.42"`. It will have a subdirectory named `"TCP:53257:80"` where all packets associated with that session will be logged.

When Snort logs in binary mode the traffic is written as it came off the wire to the log file. No conversion or translation is required, so it is much faster than any other logging mode available in Snort.

Be aware that no matter what mode you select, the log directory must exist; Snort will not create a new log directory for you just because you include it with the `-l` switch.

There is another advantage of logging in binary and that is portability. The binary logs created by Snort (or by tcpdump or any other program that supports the libpcap binary format) can be run back through Snort using the `-r` switch or any other libpcap based tool.

Logging to Directories (-K ascii)

- One file created per protocol/port pair
- What happens if someone does a full port scan?

65536 TCP ports + 65536 UDP ports = 131072
files created, possibly in a single directory!



Log Directory (/var/log/snort)

Directory name:

10.1.1.234

Files:

TCP: 22123-23

UDP: 30432-53

Directory name:

1.2.9.4

Files:

TCP:1029-80

UDP: 1056-111

Intrusion Detection In-Depth

Here's an example of how Snort's "-K ascii" logging mechanism works. Snort, by default, logs to "/var/log/snort" unless otherwise specified with the `-l name` command line switch, identifying a log directory that must already exist. When packets come in, a subdirectory in the log directory is created for the relevant source IP address if it doesn't already exist. Then files containing the packet dumps are created within that directory based on the protocol and ports.

The format for Unix host log file names is `PROTOCOL:sourceport-destport`. Take a look at the directory `1.2.9.4`; it has a file named `"TCP:1029-80"` representing a given TCP session and for TCP and `"UDP:1056:111"` representing a UDP exchange.

There is one really big drawback to using ASCII mode if you're going to be using Snort in an uncontrolled environment. It's relatively easy to have a DoS attack against the file system of your sensor when a very large number of files must be created to store the logged records.

For example, take the case of a full portscan attempting to connect to every port, each attempt logged. Soon you would have over 130,000 log files created in the logging directory. In cases where you're going to be using Snort in an uncontrolled environment for long periods of time, it's best to use the default binary logging mode.

NIDS Mode

- This is where it gets interesting!
- Tune configuration file with preprocessors, settings, and rules and start Snort
- Snort at its most complex
 - Variety of options for packet analysis and logging
- Most often deployed in NIDS mode
- Basic run consists of merely specifying a configuration file
 - c <configuration file>

```
snort -c snort.conf
```

Intrusion Detection In-Depth

Now we get to the interesting part of Snort. When Snort is in NIDS mode, it is loaded with a configuration file containing runtime directives, preprocessors, and rules, etc. When running in this mode, it is capable of analyzing network traffic in real time for conditions that will generate alerts and log the offending packets.

Most Snort deployments use the NIDS mode since this is where malicious traffic can be detected and/or blocked. Activating Snort in NIDS mode is a simple matter of specifying the -c switch on the command line with the name of the configuration file, as you know is "etc/snort.conf" found in the install directory.

As covered before, the -Tc option allows you to test your Snort configuration, looking for syntax issues or problems in the configuration file and rules, then quitting whether or not it finds errors. Use the following command:

```
snort -Tc snort.conf
```

The Snort process will die and report any errors (syntax or otherwise) that are found in your configuration or rules files. Make sure to run this after you make configuration or rules file changes, before reloading or restarting Snort.

Logging and Alerts

- Default logging/alert directory: `/var/log/snort`
Specify an alternate with `-l directoryname`
- Default alert mode: Full
`-A <mode> fast, full, none, console, unsock, cmg`
`-s` to alert to syslog
- Default logging mode: libpcap format
`-K ascii` (ASCII log format)
`-N` or `-K` (no logging, alerts still generated)
or
`config nolog` (snort.conf)

Intrusion Detection In-Depth

NIDS mode can generate log entries and/or alerts from rules that fire. An alert contains information regarding an offending packet or stream that caused a rule to fire. This includes some detail about the rule such as an accompanying description message, and a Snort ID – to name a few, along with details about the packet itself. The default mode is to capture full details about the alert in a file named "alert" created in the log directory.

Let's look at the various alert types. Fast mode prints an abbreviated message containing the IP addresses and port, timestamp, and alert message for the event on a single line. Full mode prints the alert message plus the full packet headers. Alerts can be turned off altogether using the none keyword, although logging still happens normally. Console mode prints alerts to the console and is useful mainly for debugging. The unsock option forwards alerts to a Unix socket for another listening program to process.

The cmg Chris Green vanity named alert mode also writes to the console, but gives more detail about the packet. Chris was a primary developer on Snort for many years, and he liked his alerts with packet data. This mode is great for lab/research work, however, be warned that this may become obsolete in the future.

Alert messages can be sent to syslog as well with the `-s` switch. The default syslog facility and level are LOG_AUTHPRIV and LOG_ALERT respectively.

As with running Snort in logging mode, running in NIDS mode provides two primary logging modes available from the command line. The default is libcap style binary logs, but you can use the `"-K ascii"` option for ASCII logging. The `-N` or `"-K none"` command line switches disable logging, yet still generate alerts.

If you prefer, this same option is available in "snort.conf" as "config nolog". Speaking of "snort.conf" for use in setting log configuration, you can use it set the directory using "config logdir:<dir>", yet it has no capability to alter default alert settings.

Alert Modes Types and Output

Full

```
[**] [1:1149:2] WEB-MISC count.cgi access [**]
[Classification: Attempted Information Leak] [Priority: 2]
01/23-00:50:40.355272 10.1.1.2:1971 -> 10.1.1.4:80
TCP TTL:64 TOS:0x0 ID:25791 IpLen:20 DgmLen:163 DF
***AP*** Seq: 0xD7A171BB Ack: 0x67235EB9 Win: 0x4470 TcpLen: 20
[Xref => http://www.securityfocus.com/bid/128]
[Xref => http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-1999-0021]
```

Fast

```
01/22-16:16:41.427377 [**] [1:618:1] INFO - Possible Squid Scan [**]
[Classification: Attempted Information Leak] [Priority: 2] (TCP)
10.1.1.75:37729 ->
10.26.15.218:3128 (entire alert appears on a single line)
```

Syslog

```
Sep 28 05:59:18 cerberus snort: [1:990:1] WEB-IIS_vli_inf access
[Classification: Attempted Information Leak] [Priority: 3]:
(TCP) 10.152.9.230:14609 -> 10.196.107.166:80
```

Intrusion Detection In-Depth

This slide illustrates the three primary text-based alerting modes supported. The full mode is the default mode of placing information in the alert file named “alert” about the offending packet’s header data. The message associated with the rule is output along with the Snort ID (SID) – a unique numerical designation for the rule.

The fast mode gives a more abbreviated view of the noteworthy packet, condensed to a single line ostensibly for the purpose of being processed by some other tool. The syslog mode also gives an abbreviated message, but it is stored in the default syslog file for the given Snort sensor.

The “numbers and dots” notation preceding the alert message ([X:Y:Z]) is the Snort unique event identifier. This series of metadata was developed to make Snort’s output easier to process with automated tools without depending on the Snort alert message because the metadata is a fixed format whereas the message is not. The first number signifies the “Generator ID”, the part of Snort that generated the alert. The second number is the Snort ID, the ID number of the actual rule that generated the event notification. The third number is the revision number, the version number of the rule that fired.

NIDS Mode Usage Examples

Log to a directory in ASCII:

```
snort -c snort.conf -l /var/log/snort -K ascii
```

Log in binary mode to a directory with fast alerts:

```
snort -c snort.conf -l /var/log/snort -A fast
```

Turn off logging but leave syslog alerting on:

```
snort -c snort.conf -N -s
```

Intrusion Detection In-Depth

Above are some examples of real world command line configurations for Snort.

The first example reads the "snort.conf" configuration file from the current directory and logs the output in ASCII to the directory "/var/log/snort". The /var/log/snort file is the default and unnecessary in the above configurations, but included to be more explicit.

The second example again reads the configuration file from the current directory's "snort.conf" file and records abbreviated alerts in the "alert" file found in "/var/log/snort". It also logs the output in binary mode in that same directory. The final option again reads from the configuration file "snort.conf" in the current directory, turns off conventional logging, but sends abbreviated messages to the syslog facility.

Interlude: Plug-ins

- Plug-ins are modular pieces of code written for Snort that allow programmers to extend the functionality of the program
 - Snort can be repurposed to do virtually any kind of traffic analysis task with plug-ins
- Three kinds of plug-ins:
 - Preprocessor: Packets are examined/manipulated before being handed to the detection engine
 - Detection: Perform single, simple tests on a single aspect/field of the packet
 - Output: Report results from the other plug-ins

Intrusion Detection In-Depth

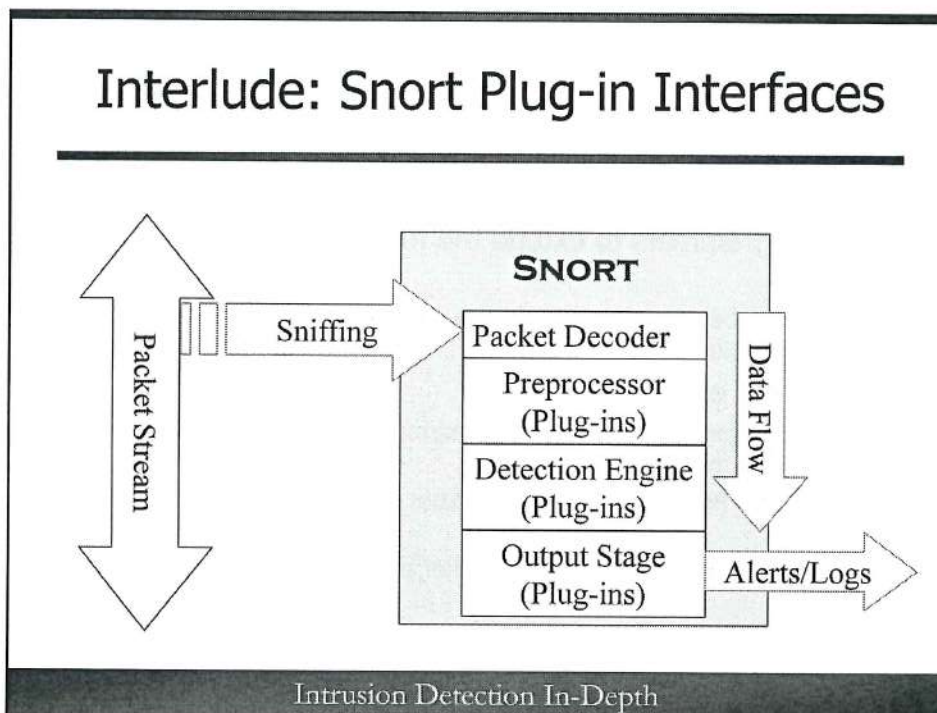
While this and the following slide seem to be misplaced in the "Running" section, they have no natural place in our category of sections. However, you need to understand the concept of Snort plug-ins because they will be mentioned. In Snort, plug-ins are code modules that allow Snort to redefine or enhance some aspect of its runtime functionality or detection capabilities. Using the plug-in system allows Snort's base functionality to be reconfigured in many interesting ways. Snort currently offers three different types of plug-ins: preprocessors, detection, and output.

Preprocessors take the decoded packets from the Snort packet decoder and can examine or manipulate them before they are handed to the detection engine. All of the preprocessors are called once per packet. This is the place where functions like portscan detection, IP defragmentation, or TCP stream assembly take place.

Detection plug-ins are used in the rules to examine a particular field of the decoded packet for certain values or patterns. They are potentially run many times per packet and therefore need to minimize their impact on the CPU as much as possible.

Output plug-ins define the way Snort presents its data. Snort generates alerts and packet logs with output formats that can be redefined easily via output plug-ins.

Interlude: Snort Plug-in Interfaces

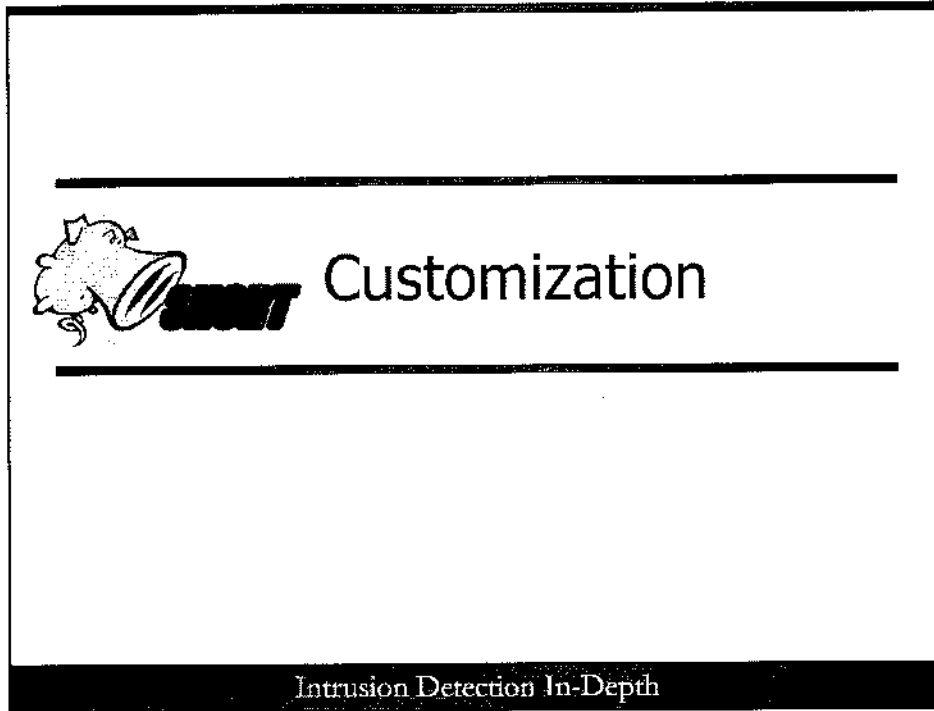


This slide shows the way packets travel through Snort and are processed by the three types of plug-ins. The packets are captured from the network interface. There is a high level packet decoder that identifies the various layers found in the packet.

The packet is then passed to preprocessor plug-ins for further processing – say for instance, reassembly of a TCP session using the stream5 preprocessor. Then, it is sent to the detection engine where it is processed against the Snort rules that contain detection plug-ins.

Detection plug-ins are somewhat less visible than the other modules in Snort because they are specified as a part of rules instead of appearing in the configuration file. There is currently a large set of detection plug-ins available in rules that we will soon discuss.

Finally, alerts are sent to the output plug-in for display or storage.



This page intentionally left blank.

Detection Scenario

- Suppose you captured some traffic from a compromised host and wanted to refine an existing Snort rule to detect any subsequent attacks
- Compromised host starts a netcat listener on TCP port 30333 with access to the command line (cmd.exe) of Windows 7 host, following Snort alert generated

```
09/17-15:53:44.597054  [**] [1:18756:4] INDICATOR-  
COMPROMISE Microsoft cmd.exe banner Windows 7/Server  
2008R2 [**] [Classification: Successful Administrator  
Privilege Gain] [Priority: 1] {TCP}  
192.168.11.24:30333 -> 184.168.221.63:48938
```

Intrusion Detection In-Depth

Here is a scenario that we'll use to present an existing Snort rule, parse its components for an understanding of the purpose of the rule, and refine it in some ways for our specific scenario and introduce important features not present in the existing Snort rule. Let's say that you are responsible for monitoring a college campus network where you have Snort installed and you are able to save full capture packets for a reasonable amount of time for retrospective analysis.

A college campus network may not have the same firewall restrictions as a corporate network just because the students', researchers', and other needs are so diverse. This particular network allows inbound traffic, such as a SYN on a listening port of a network host. Now, suppose a student's Windows 7 operating system laptop has been compromised via a phishing attack, administrator access has been attained, and a netcat listener is installed on port 30333 to provide subsequent access to that host. The netcat listener provides access to the command line (known as cmd.exe) upon connection.

You discovered the compromise via an existing Snort rule. However, you want to refine it to be more efficient or to be more specific to the traffic you witnessed. The alert is shown above.

Current Snort Rule for Windows 7

```
alert tcp $HOME_NET !21:23 -> $EXTERNAL_NET any \
(msg:"INDICATOR-COMPROMISE Microsoft cmd.exe banner \
      Windows 7/Server 2008R2"; \
flow:established; \
content:"Microsoft Windows"; depth:18; \
content:"Copyright |28|c|29| 2009"; distance:0; \
content:"Microsoft Corporation"; distance:0; \
metadata:policy balanced-ips drop, \
policy connectivity-ips drop,policy security-ips drop;\
reference:nessus,11633; \
classtype:successful-admin; \
sid:18756; rev:4;)
```

Intrusion Detection In-Depth

This is the rule that was associated with the alert that you received from the compromise. It looks somewhat daunting in its entirety, but it actually is more easily understood if we parse its components individually. We'll step through that process in the next set of slides.

As a preview and summary – this rule looks for any traffic originating from your internal network that you defined previously as "HOME_NET" from any ports other than those between 21-23 inclusive, to any host not in your network to any port. An alert generates the output defined by the **msg** keyword's argument of "INDICATOR-COMPROMISE Microsoft cmd.exe banner Windows 7/Server 2008R2". The backward slashes denote that the rule continues on the next line. You must use the backward slash when using multiple lines in a rule otherwise Snort won't understand the syntax and it will generate an error.

Next, we look for all the conditions that must be present for the rule to fire. These include finding an established session – one where there has been a successful three-way handshake. We have several different content searches that are found in specific offsets in the payload. Finally, there are various metadata identifiers – some appear in the alert to give it more context.

The **metadata** keyword itself identifies the action to take when the rule is included in a particular inline policy. Rules are placed in particular policies when created, however the user can modify the policy where the rules are placed. The "balanced" policy names a category that is intended as generic, and good for most sites especially those that do not want to do much customization. The "connectivity" policy is streamlined with rules that are almost certainly associated with malicious activity – few or no false positives exist. The "security" policy includes a broad set of rules that are likely to produce many false positives unless specifically customized by the IDS/IPS administrator for the protected site. Rules in this policy need to be refined and customized to reduce false positives. This particular rule says to drop the packets associated with the session where this activity is detected in inline/IPS mode, regardless of the policy applied to the rule. An alert will be created in IDS mode.

Before We Begin: Run the Rule By Snort

```
ipvar HOME_NET any
ipvar EXTERNAL_NET any

preprocessor stream5_global: max_tcp 8192, track_tcp yes, track_udp no, \
  track_icmp no max_active_responses 2 min_response_seconds 5
preprocessor stream5_tcp: policy windows, detect_anomalies, require_3whs 180

alert tcp $HOME_NET !21:23 -> $EXTERNAL_NET any \
(msg:"INDICATOR-COMPROMISE Microsoft cmd.exe banner Windows 7/Server \
2008R2"; flow:established; content:"Microsoft Windows"; depth:18; \
content:"Copyright |28|c|29| 2009"; distance:0; content:"Microsoft \
Corporation"; distance:0; sid:18756; rev:4;)

-----
snort -A console -K none -q -r cmdexe.pcap -c cmdexe.rule

09/17-15:53:44.597054  [**] [1:18756:4] INDICATOR-COMPROMISE Microsoft
cmd.exe banner Windows 7/Server 2008R2 [**] [Priority: 0] {TCP}
192.168.11.24:30333 -> 184.168.221.63:48938
```



Intrusion Detection In-Depth

cmdexe.rule
cmdexe.pcap

It's early in the process of learning about rules to show the rule we'll use as the foundation for understanding all the parts and options. The reason for showing the rule is to introduce you to how to run Snort and receive output. There are demo pcaps and rule configuration files supplied to allow you to run Snort as shown on the slides.

The top panel of the slide is the contents of the configuration file used in the run. This particular configuration file combines some of the variables and preprocessors normally found in "snort.conf" in live or production mode. We are executing Snort in a readback mode where we have already captured traffic and now want to analyze it using Snort. The rule that will be demonstrated has a unique Snort Identifier (SID) of 18756. It has been modified to remove some of the "metadata" because inclusion of those options requires additional files to be supplied to run Snort. We want to keep this as clean and simple as possible.

This rule has the variables \$HOME_NET and \$EXTERNAL_NET so we need to assign those values; "any" is selected to make this a generic run. Also, the use of the **flow:established** option requires the use of stream5 preprocessor settings that reassemble individual packets into a stream or session. We use some of those stream5 settings found in the "snort.conf" that comes with Snort.

Snort is run with several command line switches "-A console" puts the output on the screen "-K none" doesn't log the alert, and "-q" puts Snort into quiet mode so we don't have to deal with copious start-up messages. We read our pcap in using the "-r" and finally we include the configuration file/rule named "cmdexe.rule" with the "-c" switch.

The output contains a date and time, as well as the SID and revision numbers, the message from the rule and the source and destination IP's and ports associated with the traffic that caused the alert to fire.

The **reference** keyword identifies where more information can be found, the **classtype** is an assessment of the type of activity, the **sid** the unique Snort identifier number, and the **rev** as the revision number of the times that the rule has been altered.

Day 4 demonstration pcaps are found in `/home/sans/demo-pcaps/Day4-demos` on the VM.

Wireshark "Follow the Stream" of Compromised Session

```
Stream Content
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\judy\Desktop\netcat\netcat>dir
dir
Volume in drive C has no label.
Volume Serial Number is 3205-901E

Directory of C:\Users\judy\Desktop\netcat\netcat

05/11/2013  12:07 PM    <DIR>          .
05/11/2013  12:07 PM    <DIR>          ..
11/28/1997  01:48 PM             12,039 doexec.c
07/09/1996  03:01 PM             7,283 generic.h
11/06/1996  09:40 PM            22,784 getopt.c
11/03/1994  06:07 PM             4,765 getopt.h
02/06/1998  02:50 PM            61,780 hobbit.txt
11/28/1997  01:36 PM              544 makefile
01/03/1998  01:37 PM            59,392 nc.exe
```

Intrusion Detection In-Depth cmdexe.pcap

Here is some of the traffic collected from the compromised session as reconstructed by Wireshark's "Follow the Stream" processing.

The first two lines of output show what is displayed when the cmd.exe or command process is started by Windows. If you recall the netcat listener on the compromised host presented command line access when a connection was made. The signature looks for the content of "Microsoft Windows" along with the copyright and date and the second iteration of "Microsoft Windows". We'll look at the **depth** and **distance** keywords in the rule that position the search for more accuracy and efficiency in upcoming slides.

✦ To see the above output run:

wireshark cmdexe.pcap (select Analyze-> Follow TCP Stream)

Snort Rules

- Snort rules have a simple format that allows great flexibility in single packet analysis
 - Frequently, this is enough to pick up many attacks
 - Multi-packet events/attacks are generally detected with preprocessors
 - Snort's stream5 and frag3 preprocessors allow stateful rule-based analysis to occur

Intrusion Detection In-Depth

At the heart of Snort's detection capabilities and customizations are the Snort rules. The Snort rules syntax was developed to be simple to read, write, and understand. As Snort has matured, you may find that the rules are not so easy to read, write, or understand any more. Regardless, simple or complex rules offer the writer highly flexible and customizable options. One of the most valuable benefits of open source Snort and rules is that you can write your own. The rules are available for you inspect and alter, allowing you to tweak existing ones or create your own.

The fact that you are able to see the rules is a giant benefit. Some commercial IDS/IPS offerings keep the rules hidden so you cannot see what causes alerts to happen. This precludes your ability to determine false positives if you do not have a clue what the rule does. All you need to do to associate a Snort rule with an alert is examine the SID in the output then search the rules directory for the rule with that same SID.

For more stateful attack detection (for example, portscan detection) the preprocessor system is used instead. And, if there is a particular functionality that you'd like, but you cannot write a conventional Snort rule or there is no existing preprocessor, Snort has API's to enable you to write your own rule or preprocessor. The stream5 and frag3 preprocessors, for instance, add the capability for the Snort detection engine (and rules system) to apply to multiple reassembled packets, as well as provide a variety of protocol anomaly detection options.

What Are Rules?

- What is the difference between signatures and rules?

Mike Poor definition:

A signature is a construct to find specific patterns in traffic.

A rule is the grammatical instantiation of a particular signature.

- Rules define what Snort should watch for
- They define who and what constitutes an attack or interesting activity
- They inspect packet headers, payload or both for designated values

Intrusion Detection In-Depth

Mike Poor, instructor extraordinaire and IDS/IPS guru, makes a distinction between the terms signatures and rules. He defines a signature as a construct that defines what patterns to examine in traffic. He characterizes a rule as the instantiation of a particular signature. If you have a given signature, you can create a rule in any "language" that supports it – rules in Snort, scripts in Bro, etc.

By defining rules, you tell Snort what traffic is to be considered suspicious, overtly hostile, noteworthy, etc. Rules define everything from the "who" is involved (source and destination IP addresses) to "what" is considered hostile (a particular string in a given protocol, for example).

Rules can be written to be very specific, looking for particular payload content and packet attributes, or they can be very general, specifying only a single IP or port. This allows you great flexibility and the capability to fine tune individual rules to aid in minimizing false positives, or to focus on certain traffic attributes.

Basic Rule Anatomy

- Each rule has two parts:
 - Rule header
 - Rule options
- Specific syntax for each part
- Rule header is **required**, rule options are not
- Rules may be entered in the rules file on multiple lines using the "\" continuation character

Intrusion Detection In-Depth

An individual rule is broken into two general parts. The first part, the rule header, defines "who" must be involved in order for the traffic to be considered by the rule options. The second part (the rule options) defines the "what" must be involved. This includes packet header information or the contents of the payload.

Generally speaking, both sections are used for most rules. It is possible to create rules with only a rule header so that the same action can be taken for the provided hosts and ports. This is usually the case where pass action rules are used to ignore traffic between given hosts and ports.

Rules can be written on multiple lines in the associated rules file using the "\" continuation character at the end of a rule line.

Rule Header

```
alert tcp $HOME_NET !21:23 -> $EXTERNAL_NET any
```

- Defines who is involved:
 - Action
 - Protocol
 - Source and destination IPs
 - Source and destination ports
 - Direction of traffic

Intrusion Detection In-Depth

The rule header is the first part of each rule used to define the network protocol and the "who" must be involved. Each of the individual fields have many options (with specific syntax) that can be used to assign single values, ranges, or groups of values. Each field will be discussed in upcoming slides, covering possible values and the proper syntax for each.

Snort's detection engine breaks the comparison of a packet into two parts, corresponding to the parts of a rule. The first comparison is the rule header against the packet. If the packet does not fit the profile of one of the rule headers in a rule set, the detection engine moves on to the next packet. If the packet does fit the profile of the rule header, then the detection engine continues on to test the rule options. As such, more specific rule headers optimize performance early in the inspection process by eliminating all traffic that does not apply. This means that the more resource intense process of pattern matching is never performed.

Rule Header: Action – Tell Snort What to Do

```
alert tcp $HOME_NET !21:23 -> $EXTERNAL_NET any
```

Action Field

alert: alert and log packet when triggered

log: log only when triggered

activate: activates another rule

dynamic: dynamically enabled by activate rule

pass: ignore this traffic

drop: block and log the packet

reject: block, log, and send either TCP reset or ICMP unreachable (UDP)

sdrop: block, but do not log

inline

The default order is:

activate->dynamic->pass->drop->sdrop->reject->alert->log

Intrusion Detection In-Depth

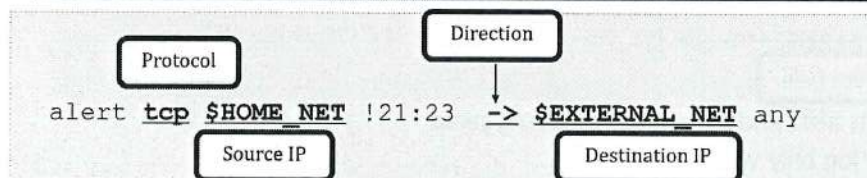
The first field in the rule header is the action field that instructs Snort what to do if the rule is triggered. The alert value instructs Snort to create an entry in the "alert" file of the log directory and to log the packet as well. The "alert" file is a single file that contains all the rule detections. The information written to this file in the default alert mode consists only of the packet header information and any metadata fields and values that you add to the rule that generate output such as the SID, priority, and message. The "log" value instructs Snort to log the packet only; it will not be recorded in the "alert" file.

The options "activate" and "dynamic" are used for coupled rules and rule types. An activated rule is one that triggers a dynamic one when the activate conditions match the traffic. The "pass" action causes Snort to ignore the packet and do no further processing of it. This is an appropriate action option when you have a vulnerability scanner in your network and wish to ignore traffic from it. Although that makes it an ideal attack target since a successful attack will cause the IDS to be blind to traffic originating from it. A different set of actions is available for inline mode. The "drop" blocks and logs the traffic, and the "sdrop" drops without logging. The "reject" mode blocks, logs, and attempts to terminate the TCP or UDP session associated with the blocked packet(s).

So you see you have a choice to refine the action for a particular rule. In other words, the rule action can override the generic configured mode for alerting and logging. For instance, you may choose to log a given rule but generate no alerts even though the command line designates logging and alerting.

Much like defining firewall rules, when working with Snort rules, correct rule ordering for accurate evaluation is very important. By default, rules are processed as follows - activate->dynamic->pass->drop->sdrop->reject->alert->log. There are command line options to supply to Snort that alter the default order. They are "--alert-before-pass" to have alerts ordered before pass rules, "--treat-drop-as-alert" to alter the behavior of drop, sdrop, and reject action to alert rather than drop, and "--treat-drop-as-ignore" that ignores rules with drop, sdrop, and reject when not in inline mode. This occurs when an inline configuration is used to run in passive mode – such as in testing. There is a "snort.conf" option "config order:" that permits you to alter the order too. Values supplied to it represent the order you desire.

Rule Header: Protocol, Traffic Direction and IP Address



- Protocol field: tells Snort what type of network traffic the rule applies to valid values include:
tcp, udp, icmp, ip
- Source IP value: where packet(s) is coming from
- Destination IP: where packet(s) is going
- Traffic direction options:
 - > defines source to destination
 - <> direction doesn't matter (bi-directional)

Intrusion Detection In-Depth

The protocol field in the rule header tells Snort what type of network traffic is applicable. Snort currently supports four different types of network traffic: IP, TCP, UDP and ICMP. Additional protocols may be added in the future.

There is a rule options keyword "ip_proto" that expands the types of protocols that may be examined. This examination is performed later in the assessment of packet contents. The limitation on the protocol values in the rule header is due to early optimizations made that group rules by the more common protocols. Packets with protocols other than TCP, UDP, or ICMP will fall into the general IP group. The ip_proto value is evaluated later when the rule options are examined.

The source IP address specifies where the traffic is coming from and the destination IP address specifies where the traffic is going to. It is possible to define the IP as something as general as a subnet, or as limited as an individual IP address. It is also possible to set multiple addresses or subnets as the source, when needed, using a special syntax called an IP List.

The traffic direction field allows you to indicate packet direction. Two options are available, allowing you to denote a specific direction of flow, or indicate that direction doesn't matter. The label "source IP" and "destination IP" are not used to denote client and server – simply the sender and receiver. This same direction option is used both for sender -> receiver or receiver -> sender.

Rule Header: IP Address Formats/Variables

```
Host:                192.168.5.5
Address/Netmask:     192.168.5.0/24
List of IPs:         [192.168.5.1,172.16.0.0/16]
Any IP address:      any
Protected IPv4 network: ipvar HOME_NET 192.168.0.0/16
Protected IPv6 network: ipvar HOME_IPV6 fe80:6f8:900:0:0:0:0:0/48
Negation:            !
Unprotected IPv4 network: ipvar EXTERNAL_NET !$HOME_NET
Unprotected IPv6 network: ipvar EXTERNAL_IPV6 !$HOME_IPV6
```

Intrusion Detection In-Depth

This slide shows the supported formats to assign the IP address including a single IP address, a subnet address with a network mask, a list of IP addresses/subnets, the keyword "any" to represent any IP address, and negation using "!". A very specific format is required to define a list of IP addresses – a comma separated list of IP addresses enclosed in brackets. Do not leave any spaces when using this format. Snort associates the reserved variable names "HOME_NET" and "EXTERNAL_NET" with the protected network IPv4 addresses and any IPv4 address except the protected network. The respective variable names are "HOME_IPV6" and "EXTERNAL_IPV6" for IPv6 networks.

The "ipvar" keyword means that the variable content contains IP address(es), for instance "HOME_NET". A "\$" prefaces a variable name to later reference a defined variable. This is true for any variable type – not just ipvar variables. You see that the variable "EXTERNAL_NET" is defined to be not the \$HOME_NET, where the "\$" refers to the value in the predefined variable "HOME_NET".

Rule Header: Port

```
alert tcp $HOME_NET !21:23 -> $EXTERNAL_NET any
```

Source Port

Destination Port

- Source port value defines originating port
- Destination port value defines receiving port
- Specified as a number or range
- Or as a list using portvar

Intrusion Detection In-Depth

The source and destination port field reference the sending and receiving ports respectively. Ports can be denoted as a single port, range of ports, list of ports, or the keyword 'any' that represents all possible ports.

Another point that needs to be made is in reference to the ICMP protocol. As you know, the ICMP protocol does not use ports like TCP and UDP. Since the rule syntax requires ports to be designated for both the source and destination, they must still be included for rules that apply to the ICMP protocol. Snort will ignore the value entered for the ports for ICMP rules, but it is traditional to specify the keyword "any".

Rule Header: Port Formats

static port:	21
all ports:	any
range:	33000:34000
negation:	!80
less than or equal:	:1023
greater than or equal:	1024:
list of ports:	portvar MISC [555,80:89,240]

Intrusion Detection In-Depth

This slide shows some of the many ways you can define the port value(s). You can assign a specific port or port ranges. It's even possible to use negation (but not with the "any" parameter) to assign all ports except the one or range specified.

When using ranges, the port numbers are inclusive. For instance, the range "33000:34000" in the example covers from 33,000 to 34,000 inclusive.

There is a portvar keyword to create a variable name that contains a port number(s). We use portvar to assign a variable named "MISC" a list of ports. This list can include one or more combinations of any of the above port designation options. As with the ipvar list syntax, make sure that there are no spaces in the list.

Defining the Ports in Our Rule

```
alert tcp $HOME_NET !21:23 -> $EXTERNAL_NET any
```

Not ports 21-23
inclusive

Any port

```
alert tcp $HOME_NET 30333 -> $EXTERNAL_NET any
```

Port 30333 only

```
portvar CMD_EXE [:21,23:]
```

```
alert tcp $HOME_NET $CMD_EXE -> $EXTERNAL_NET any
```

Any port <= 21
and >= 23

Intrusion Detection In-Depth

Examining the Snort rule that triggered the alert, we find that the source ports are specified as not ports 21-23 inclusive. That is because these are the ports that legitimately receive a command shell when a connection is made. Port 21 is ftp command mode, port 22 is secure shell (ssh), and port 23 is telnet. While ftp may be found offered on a Windows server, clear text telnet is pretty much obsolete or it should be in favor of the encrypted secure ssh. It seems foolish to include port 22 ssh since its intended encrypted payload precludes inspection of the clear text content. Our destination port is "any" because in this particular scenario, it represents the client's ephemeral port that, as you know, is typically above 1023, although any value can be assigned to it by someone creating their own TCP traffic, avoiding the use of the TCP/IP stack to open a port above 1023.

Suppose we wanted to limit our alerts to traffic with a source port of 30333 as we did in the second rule. Perhaps you discovered multiple compromises and all of them used port 30333. This is certainly more efficient for Snort to find, however it may result in a false negative if so restricted.

Finally, let's define the original rule using a different syntax that avoids the use of negative logic of the first rule. We define a portvar named CMD_EXE that includes those ports less than or equal to 21 and another that includes ports that are greater than or equal to 23. We omit port 22 this time because that is ssh and inspection of it is wasteful assuming it is encrypted. This could potentially lead to a false negative if someone were to run a rogue clear text session over port 22, but we'll take our chances for an improvement of efficiency. We use the variable \$CMD_EXE in our rule to now identify the source ports.

Rule Options

```
msg:"INDICATOR-COMPROMISE Microsoft cmd.exe banner \  
    Windows 7/Server 2008R2"; \  
flow:established; \  
content:"Microsoft Windows"; depth:18; \  
content:"Copyright |28|c|29| 2009"; distance:0; \  
content:"Microsoft Corporation"; distance:0; \  
metadata:policy balanced-ips drop, \  
policy connectivity-ips drop,policy security-ips drop;\  
reference:nessus,11633; \  
classtype:successful-admin; \  
sid:18756; rev:4;)
```

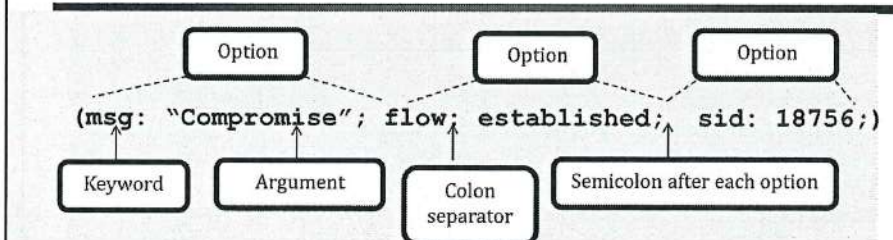
- Defines what is involved
- What packet attributes must be inspected
- Includes metadata to be placed in output alert for more context

Intrusion Detection In-Depth

The rule options portion represents the remainder of a rule. The rule options define the "what" portion of a rule – what attributes of a packet must be inspected and the values they must contain for the rule to fire. This portion of a rule is applied only if the packet meets the specifications of the rule header parameters first. If a packet has met both the requirements of the rule header and the rule options, the rule action is triggered.

Rule options can contain metadata as we'll discuss. These options provide more data and context on the alert output such as references to learn about the attack as well as an assessment of the type of attack it is. These metadata options are not used for matching payload.

Rule Options Syntax



- Rule options made up of one or more options consisting of:
 - Parentheses: must be enclosed between left and right parentheses
 - Keyword: reserved value to identify option
 - Colon: separator between keyword and argument
 - Argument: value of keyword
 - Semicolon: separates individual options and must close options list

Intrusion Detection In-Depth

The rule options section between the left and right parentheses uses reserved keywords and possibly associated arguments. Just about every keyword requires an argument value, however there are a few that do not, such as **sameip** that looks for the same source and destination IPs in the rule header. The keyword and argument are separated by a colon and individual options are separated by a semicolon.

Each keyword/argument pair comprises an option. Make sure that you close the set of specified options with a semicolon too; one of the most common mistakes is to omit this, causing an error.

Think of all the specifications in a rule, both the header and options sections, as a long "and" statement where all conditions must be met for the rule to trigger. Unfortunately, there is no way to specify "or" options conditions in a single rule. For instance, if you wanted a rule to fire if the content had either "foo" or "bar", you would have to write two separate rules.

If you think about it some options in the rules header allow for an "or" condition by the use of lists of either IP addresses or port numbers. Yet, this concept does not carry over to the rules options section.

Rule Options Types

- Rule options are driven by detection plug-ins
- We'll discuss some of the most common
 - General
 - Non-Payload
 - Payload

Intrusion Detection In-Depth

The rule options in Snort represent the available detection plug-ins. In fact, there are so many rule option keywords available that for the purposes of timeliness we will cover only a small subset of the most common options. The rule options in Snort can be broken down into several groups. We'll cover some general, non-payload, and payload options types only.

The general options provide information about the rule that may be seen in the output to better describe attributes associated with the rule, and included in the alert, like the message, Snort ID, references, priority, and classification to name a few. These can be thought of as metadata options – data about data.

There are non-payload options that generally examine fields in the packet headers. They may also examine flow direction and associated flows. Next there are options that indicate what to examine in the packet payload content, values or conditions that identify noteworthy traffic. By far, payload options outnumber all other categories – well, because that is really the purpose of the rules – to inspect the payload content for something of interest.

General (Metadata) Options

msg: descriptive message that appears in alert
msg: "WEB-MISC ftp.pl access";

sid: Snort ID that is a unique numeric value assign to every rule
sid: 1107;

gid: Generator ID that indicates what part of Snort generated the alert
gid: 1;

rev: revision number
rev: 5;

reference: attach an external reference
reference: bugtraq,1471;

classtype: classification of class of attack
classtype: web-application-activity;

priority: severity
priority: 3; # value of 1 is highest, 4 is lowest

metadata: includes more data about the rule
metadata:policy balanced-ips drop

Intrusion Detection In-Depth

General options are used to attach information to a rule and associated alert. This information or metadata is not used in the actual detection; its purpose is to help the analyst by providing some descriptive data along with the alert.

The **msg** option allows you to attach a pertinent descriptive message to an alert. Obviously, this assists in providing a relevant description when you examine the alerts. You can assign any description that you choose; just remember that if you are sharing Snort duties with a co-worker, make sure that this description is universally coherent - something that everyone can make sense of, not just you.

The **sid** is the Snort ID to easily identify a unique Snort rule. All of the rules must have a **sid** and they all must be unique. The range is:

<100	Reserved for use by the Snort team
100-999,999	Assigned by the Snort team for signatures in wide distribution
>=1,000,000	Local site defined rules

The **gid** is the generator ID that informs you what part of Snort generated the alert. By default, a **gid** value is 1, unless otherwise assigned. This means that the source of the alert is the rules subsystem. All of the **gid**'s can be found in the "snort/etc" subdirectory, in a file named "gen-msg.map". Usually, the **gid** value is not reassigned.

The **rev** or revision ID is used to track the revision count of a rule.

Metadata in Our Rule

```
msg:"INDICATOR-COMPROMISE Microsoft cmd.exe banner \  
Windows 7/Server 2008R2"; \  
flow:established; \  
content:"Microsoft Windows"; depth:18; \  
content:"Copyright |28|c|29| 2009"; distance:0; \  
content:"Microsoft Corporation"; distance:0; \  
metadata:policy balanced-ips drop, \  
policy connectivity-ips drop,policy security-ips drop;\   
reference:nessus,11633; \  
classtype:successful-admin; \  
sid:18756; rev:4;)
```

Intrusion Detection In-Depth

The rule that fired has several metadata arguments and keywords. The `msg` value appears in the alert output as a terse description, including a likeness to the associated rules file name – "INDICTATOR-COMPROMISE", followed by an explanation of the type of activity discovered. As well, there is the metadata keyword that specifies that if the rule is included in any of the policies of "balanced-ips", "connectivity-ips", and "security-ips" signifying that an IPS should apply the drop action on any traffic applicable to the rule.

There is a reference associated with a nessus rule identifier, a classification type that identifies the traffic is reflective of successful administrator access, a Snort ID of 18756, and a revision number of 4.

Non-Payload flow

- Examines direction and state of traffic flow
- Most common state option:
established: Stateful inspection has marked session as established
- Direction options:
to_server = **from_client**
to_client = **from_server**
- Configuration file stream5 preprocessor defines parameters for reassembly
 - stream5_global global settings for TCP, UDP, ICMP
 - stream5_tcp TCP specific settings including the port numbers of traffic that will be reassembled

Intrusion Detection In-Depth

Let's look at the **flow** option in more depth since it is important and often used. Rules can be written to take advantage of the stateful inspection reassembly capabilities available in Snort with the use of the stream5 preprocessor. The **flow** option requires the inclusion of the stream5 preprocessor in the configuration file. There are some general types of options that are available for the **flow** keyword; we'll cover only state options and direction options.

The state options apply to established or unestablished/stateless TCP or UDP sessions. Established means after the three-way handshake for TCP or as a series of packets using the same source and destination IPs and ports for UDP. These are sessions that have gone through Snort's stream5 processor that are reassembled as a stream. There is a "stateless" flow keyword that applies to individual packets, however no current active rules use it.

The direction options allow the user to designate the source or destination from a client/server perspective. There are four available values, although only two are unique. The **to_server/from_client** are the same, as are the **to_client/from_server** options to give the user choices to define the traffic flow as they perceive it. This makes inspection more efficient when a single side only of the conversation is examined.

The **flow** option is a very important one associated with TCP and UDP traffic. As mentioned previously, Snort processes traffic in terms of packets. But, as you know a TCP session is comprised of many different packets containing TCP segments. Content in TCP sessions is found in a segment after the TCP session has been established. UDP rules generally use **flow** to indicate the direction - not the state.

As mentioned, the stream5 preprocessor is responsible for reassembling individual related packets into a stream. There are two parts to the stream5 preprocessor – one is the stream5_global global parameters that pertain to UDP and ICMP packet reassembly. The second part is the stream5_tcp preprocessor that defines TCP settings

including the very important, and often overlooked, values of TCP ports to reassemble. Snort tries to be more efficient by reassembling traffic to ports defined in stream5_tcp. There is an option to define client, server, or both ports where traffic is examined **only** if it destined to a listed client port, a listed server port, or both client and server traffic to the listed port. It is important to understand that Snort does **not** automatically reassemble traffic to all ports. This means that TCP sessions over unusual ports will not be reassembled unless designated in stream5_tcp ports listing.

Non-Payload flow Rules

```
alert tcp $HOME_NET !21:23 -> $EXTERNAL_NET any
flow:established; \
content:"Microsoft Windows"; depth:18; \
content:"Copyright |28|c|29| 2009"; distance:0; \
content:"Microsoft Corporation"; distance:0; \
```

Payload in an established session, flow either to or from client/server

```
alert tcp $HOME_NET !21:23 -> $EXTERNAL_NET any
flow:established, from_server; \
content:"Microsoft Windows"; depth:18; \
content:"Copyright |28|c|29| 2009"; distance:0; \
content:"Microsoft Corporation"; distance:0; \
```

Payload in an established session, flow from server/to client

Intrusion Detection In-Depth

Here are two different examples of **flow**. The first uses the "established" keyword alone as found in the original Snort rule. This means that the payload is inspected in either direction – going to or coming from the client/server.

Why does this signature examine traffic in both directions when logically we think of the attack scenario as a compromised server on the protected network where the attacking client connects remotely to obtain a shell? There is also the option of an attacker establishing a reverse shell that is set up on a compromised host on the protected network to connect outbound to the attacker's host "shoveling" the shell to the attacker's host. That is when you would see the artifacts of a compromised "client" on your network accessing the shell on the attacker's "server". A reverse shell is typically used when inbound access to the compromised host is blocked.

Suppose we were just interested in protecting the scenario where no reverse shell connection would be possible; we could make the rule more efficient by restricting the flow to the direction from the server to the client. In this configuration, we would qualify the **flow** as **from_server** or **to_client**.

Just a reminder - the backslash character "\" is used to indicate that the rule continues on the next line.

Note: The 2.9.6 Snort User's Manual description of **flow** erroneously indicates that **flow** is used with TCP only. At one time this was true; however it currently is used with UDP as well. The distributed set of rules contains ones with UDP and **flow**. The best place for the most current documentation is in the "snort/doc" directory README files. These are written by the Snort developers and are very thorough and usually current.

The User's Manual is a translated version of README file descriptions provided by the developer. At times the changes do not may their way to the User's Manual or do not make their way accurately.

Payload Options content

- Examine the packet payload:
 - Content can consist of straight text, hex data, or a mix of both
 - Hex data is surrounded by vertical pipes: "|90 90|"
- Content matching may also be done by exception with a "!" character
- Optional content modifiers are available
- Multiple content options (with modifiers) may be defined

Intrusion Detection In-Depth

The third rule category is payload that has many options that examine a specific aspect of the packet payload. We'll cover some of the more commonly used ones.

The **content** keyword is the most common option that is used to examine the payload for character strings or byte values. Content inspection is both extremely powerful and computationally expensive, if not used properly. Care must be taken when writing rules using **content** options to be as precise as possible in terms of content value and the location of the content in the payload.

The payload can be examined for text data or hexadecimal data. When needed, hexadecimal and character strings can be combined into the same **content** option.

It is also possible to use the negation operator with the **content** option by adding the "!" character in front of the pattern, the option will match when the payload does not contain the pattern. It is best to use the negation **content** operator in conjunction with other **content** keyword value pairs that search for the existence of some other known content.

There are multiple modifiers available for use with the **content** option. These modifiers are useful for improved tuning and optimization of content matches. You can define multiple **content** options (with modifiers) in a single rule. Think of these as "and" conditions for content. We will cover a subset of these.

Payload content Rules

Plain text

```
content:"Microsoft Windows";
```

Mixed plain text and hex

```
content:"Copyright |28|c|29| 2009";
```

Multiple content checks

```
content:"Microsoft Windows"; depth:18; \
content:"Copyright |28|c|29| 2009"; distance:0; \
content:"Microsoft Corporation"; distance:0; \
```

Content with exception matchings

```
content:"Microsoft Windows"; depth:18; \
content:"Copyright |28|c|29|"; distance:0; \
content: !"2010"; distance: 0\
content:"Microsoft Corporation"; distance:0; \
```

Intrusion Detection In-Depth

cmdexe.rule exception.rule
cmdexe.pcap

Here are some sample rules that use the **content** option. The first sample rule has a simple **content** match, looking for the **content** of "Microsoft Windows".

The second example rule has mixed hex and text data to represent "Copyright (c)" in the **content** argument field. The parentheses are represented in hex because they have another purpose in Snort syntax to enclose the rule options. Note that the hex data is enclosed in "|" (better known as the "pipe" character), to indicate to the parser that this is hex data. When checking byte code in hex, Snort ignores white space used in the rule content. Mixed hex and plaintext data is considered sequentially, the pattern matcher in Snort doesn't break the data up into separate matches.

The third example rule shows a situation where three **content** matches are being performed. These pattern matches are performed separately from each other and may occur anywhere in the packet. The order in which the **content** is placed in the rule doesn't necessarily correspond to the order in which they are found in the payload.

The final sample rule shows the exception matching feature looking for content that does not contain the string "2010". This negative or exception **content** match should not be overused since it is computationally expensive. Snort has search optimizations it employs for finding content in the payload. It is not possible to use these optimizations when an exception match is used without other non-exception content matches. If at all possible it should have another search restricting criteria such as existing content – like one or more **content** searches. Otherwise, the pattern matcher must inspect payload in every packet that matches the associated rule header values.

✦ To run the rules seen above, execute the following:

```
snort -A console -K none -q -r cmdexe.pcap -c cmdexe.rule (pertains to the first three rules)
```

```
snort -A console -K none -q -r cmdexe.pcap -c exception.rule (pertains to the last fourth rule only)
```

Payload content Modifier offset

- Defines offset from the beginning of the payload to start content matching
- Default offset is zero (first byte of packet payload)
- Limits the amount of data that must be searched per packet; improves performance

```
content:"Microsoft Windows"; depth:18; \
```

Original modifier

```
content:"Microsoft Windows"; offset:0; depth:18 \
```

Explicit modifier

Intrusion Detection In-Depth

Let's examine some **content** modifier keywords. A **content** modifier keyword and value must come directly after the associated content.

The **offset** option defines the point in the packet payload to begin searching for a content match. Note that the content matcher begins counting at the zero offset from the beginning of the payload as the default. Therefore, it was not necessary to explicitly indicate an offset. However, you can supply it as shown in the second rule if you want to be more obvious about it.

The **offset** option is extremely useful for localizing particular expected content in a packet as well as speeding up the pattern matcher by reducing the amount of data to be searched in a particular packet.

The **offset** value ranges from -65535 to 65535. You may be wondering when you would ever use a negative offset. Actually, none of the current rules uses a negative offset so while the option is provided, it does not appear to be very useful at this time.

Payload content Modifier depth

- Limits the depth from the initial offset of a content search
- If no offset is specified, the depth is set from the beginning of the packet payload
- Once again, limits amount of data that must be searched per packet, improves performance

```
content:"Microsoft Windows"; depth:18; \
```

Intrusion Detection In-Depth

The **depth** option limits the number of bytes from the starting offset pointer that will be searched. If no **offset** is specified, the depth distance is from the 0 byte offset of the beginning of the payload. Using **offset** and **depth** in concert allows rules to be written to examine very specific areas of the packet for precise values, speeding up the overall performance by reducing the amount of data to be searched.

The **depth** keyword represents a byte count. Unlike **offset**, it must have a positive value of 1-65535 since it must identify the number of bytes to search. For instance let's say that we have a payload of "ABCDE...P" in our content. Now, suppose we want to make sure that "CD" appears exactly as it falls in this string. We would specify the following: **content: "CD"; offset:2, depth:2**. The **offset** counting starts at 0 and the "CD" requires 2 bytes.

Our rule defines that the **content** of "Microsoft Windows" must be found wholly within the first 18 bytes as an offset of 0 is implied. There may some confusion about when the counting starts at 0 or 1. **Offset** starts counting at 0; **depth** starts counting a 1 since you are counting actual existing bytes, not a placement pointer for the beginning of the count.

Payload content Modifiers distance and within

- "distance" limits the relative offset from previous content match
- "within" limits the relative depth from previous content match
- Once again, limits amount of data that must be searched per packet, improves performance

```
content:"Microsoft Windows"; depth:18; \  
content:"Copyright [28|c|29] 2009"; distance:0; \  
-----
```

Original modifier

```
content:"Microsoft Windows"; depth:18; \  
content:"Copyright [28|c|29] 2009"; within:50; \  
-----
```

More restrictive modifier

Intrusion Detection In-Depth

cmdexe.rule within.rule
cmdexe.pcap

The content modifiers **distance** and **within** behave the same as **offset** and **depth**; except, instead of relative to the beginning of the payload, they are relative to the location of an assumed previous content match. The first rule specifies that the copyright content search must begin immediately after the content of "Microsoft Windows" and ends either when it is found or when the remaining payload is search if not found. There is no restriction how far to search into the payload - usually since it is variable or unknown.

We can put a restriction on the number of bytes examined if, for instance, we know that the copyright content was wholly contained **within** 50 bytes of the end of the string "Microsoft Windows". We use **within** to accomplish this; this has an implied distance value of 0.

✦ To run the rules seen above, execute the following:

```
snort -A console -K none -q -r cmdexc.pcap -c cmdexe.rule (pertains to the first rule)
```

```
snort -A console -K none -q -r cmdexc.pcap -c within.rule (pertains to the second rule)
```

Payload content Modifier `fast_pattern`

- Snort gains efficiency using pattern matching algorithm
 - Default pattern matching uses longest content value to match
 - May not always be what you want or most efficient
- **`fast_pattern`** is a content keyword modifier that offers other matching options/conditions

Intrusion Detection In-Depth

An exhaustive function in terms of time and necessary processing is examining a reassembled stream payload or a single packet payload to find the desired content. Snort uses a default pattern matching algorithm to make this process as streamlined as possible. The expected behavior of the pattern matcher is to take the first, longest, non-negative (doesn't contain the "!" before content value) content value and use it in the pattern matching phase.

One reason to use the **`fast_pattern`** content modifier is when the longest content match for a given protocol is found in many, or all packets associated with the protocol. The example given in the Snort User's Manual is a string of "INVITE sip|3A|" is frequently found in all Session Initiation Protocol (SIP) traffic. Suppose you write a SIP rule that includes that content as well as another content string that has fewer bytes. It is more efficient for the pattern matcher to search for the other shorter string first because it is more unique. Otherwise, the pattern matcher examines every SIP packet to find the second string. Less processing is performed when the smaller string is sought and when found, the packet is then searched for "INVITE sip|3A|".

Payload content Modifier fast_pattern Rule

- Use fast_pattern modified content for more efficient pattern matching

```
content: "netcat"; fast_pattern; \  
content: "Microsoft Windows"; depth:18; \  
content: "Copyright |28|c|29| 2009"; distance:0; \  
content: "Microsoft Corporation"; distance:0; \  
  
snort -A console -K none -q -r cmdexe.pcap -c fast-pattern.rule
```



Assume that we wanted to alter this rule so an alert is triggered when the **content** of "netcat" is found in the associated traffic. As you can imagine the existence of "netcat" in the payload is likely more unique than the **content** found in the existing rule so that should make it a more efficient rule since it optimizes the pattern matcher as fewer packets/streams will be scrutinized. However, we need to use the qualifier of fast_pattern otherwise the longest **content** of "Microsoft Corporation" will be selected as the first content to be used by the pattern matcher.



To run the rule seen above, execute the following:

```
snort -A console -K none -q -r cmdexe.pcap -c fast-pattern.rule
```

Payload pcre

- Adds the power of Perl Compatible Regular Expressions to Snort rules language
- Below rule content check looks for the anchor and pcre content of "Copyright (c)" along with a more generic pcre match for the representation of the year

```
flow:established; content:"Microsoft Windows"; depth:18;  
content:"Copyright |28|c|29| "; pcre:"/Copyright \(c\) 2\d\d\d/";  
content:"Microsoft Corporation"; distance:0; sid:54322;)  
  
snort -A console -q -K none -r cmdexe.pcap -c pcre.rule
```



Regular expressions are extremely powerful, and very flexible. Traditionally they have been rather slow for use in IDS/IPS, but with some optimizations, **pcre** was ported into Snort without significantly affecting performance. One optimization that you can supply is when using **pcre**, if at all possible, anchor the **pcre** expression with a **content** option keyword containing a value with the largest string or bytes of non-changeable content searched by the **pcre** expression. The pattern matcher cannot be applied to **pcre** content because it does not understand regular expression syntax.

For instance, in the above rule, you see that there is an anchor **content** containing "Copyright |28|c|29|" (Copyright (c)) before the **pcre** expression. This is the fixed part of the string that does not change. This allows the pattern matcher to efficiently find the content and then the less efficient processing will examine the specifics of the **pcre** content containing that same string.

The original rule contains an exact content of "2009" to identify the year associated with Windows 7/Server 2008R2. Yet, this is restrictive and there were several similar Snort rules for different operating system versions of Microsoft. We can make this rule more generic to cover existing and future versions of operating systems without creating a new rule each time a new operating system is released with a year other than 2009. First, we express the non-changing part of the **content** "Copyright(c)". Next we repeat this same content in **pcre** expression language that requires us to use an escape backslash character for the parenthesis denoting the copyright symbol. That's because parentheses have a particular meaning in regular expressions – not the text context we need.

Next we use regular expression syntax to give us flexibility with the year associated with the banner message. We begin the year with the value of "2" followed by a series of 3 digits "/d". This is an easy fix if you are familiar with regular expressions, although learning regular expression syntax can take some time.

Regular expressions are quite complex and a thorough discussion of them is out of the scope of this class. For tutorials on regular expressions, check the book: Mastering Regular Expressions, published by O'Reilly, as well as the following websites:

<http://www.zvon.org/comp/r/tut-Regexp.html>

<http://www.php.net/pcre>

✦ To run the rules seen on the slide, execute the following:
snort -A console -K none -q -r cmdexe.pcap -c pcre.rule

Payload isdataat

- Examine payload for existence of data at an offset/relative byte number

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 143 \
(msg:"PROTOCOL-IMAP DIGEST-MD5 authentication method buffer \
overflow"; flow:established,to_server; content:"AUTHENTICATE \
DIGEST-MD5"; nocase; content:"|0A|"; within:2; \
isdataat:256,relative; content:!"|0D 0A|"; within:256;sid:123456;)

snort -A console -K none -q -c local-imap.rule -r imap.pcap
02/06-21:17:40.512285  [**] [1:123456:0] PROTOCOL-IMAP DIGEST-MD5
authentication method buffer overflow [**] [Priority: 0]
(TCP) 10.0.2.101:55032 -> 10.0.1.102:143
```



Intrusion Detection In-Depth

local-imap.rule
imap.pcap

The **isdataat** payload option is another useful feature to detect the existence of data a fixed or relative offset. One of the most common uses for this option is for buffer overflow detection. Take a look at the above rule. It has two initial **content** matches followed by **isdataat:256,relative**. This examines the payload 256 bytes following the match of the second **content** search or 0x0a. If this data exists and no carriage return/line feed, also known as CRLF, (0x0d 0a) is found, it indicates that more data is present than expected.

The IMAP protocol needs user input of an authentication string comprised of a username and digest to identify the user. IMAP is a line-oriented protocol where the CRLF signifies the end of input. Therefore if no end of input is detected within 256 bytes after the user input, a buffer overflow is likely.

There is another common payload option **nocase** in this rule that make the search case insensitive.



To run the rule seen above, execute the following:

```
snort -A console -K none -q -r imap.pcap -c local-imap.rule
```

My New Rule Doesn't Work! What Now?

- Give up, panic, or **debug**
- Three possible reasons:
 - The rule is wrong
 - Snort is not properly configured
 - The traffic/pcap you are using has an issue
- Most common reason is the rule is wrong
 - Debug it option by option
 - Test the header option(s) only
 - Remove each rule option, one by one, to see if the alert fires

Intrusion Detection In-Depth

You or someone else has just written a new rule, tested it against a pcap you have that you believe is representative of the issue for which you wrote the rule. And, sadly, you get no alert. Your instinct is to immediately blame the oversight on a bad rule or maybe a bad rule writer. Most of the time, this is where the problem lies.

However, don't discount some other explanations – a configuration file that does not handle the type of traffic you are using or even a pcap that is corrupted in some way. You may not have the required preprocessor loaded in your "snort.conf", for instance the "frag3" preprocessor handles fragmentation. If your pcap has fragmented packets, Snort will never assemble the fragments so the payload content you seek may not be found. Also, if your \$HOME_NET and \$EXTERNAL_NET variables are not properly set, you will receive no alert. A final check is to make sure that the file location of the new rule is included in the Snort configuration file.

As far as a corrupted pcap, make sure that there are no checksum errors. By default, Snort drops packets with any incorrect checksum. After all, that is what the destination host does. Snort can be easily evaded if it does not behave in the same manner as the receiving host. Or, it is possible that the snaplen – the default number of bytes captured for each packet – is insufficient and the entire packet was not captured.

Rule debugging is a methodical iterative process. First, delete all the rule options except for the **msg** and **sid**, leaving only the header part of the rule. It's possible that something is wrong in the header and is easily corrected. The rule options are next in the debugging process. Delete them all (except **sid** and **msg**) and add them back one by one until the alert no longer triggers. Investigate that option – maybe the value supplied to the keyword is incorrect – a bad offset or distance value so tweak those until the rule fires. Or maybe the **content** is both upper and lower case, however you haven't qualified it with **nocase**.

You'll get to do some debugging now if you begin the Snort exercise that is referenced on the next slide.

What's Wrong with This snort.conf?

Workbook

Exercise: "What's Wrong with this snort.conf?"

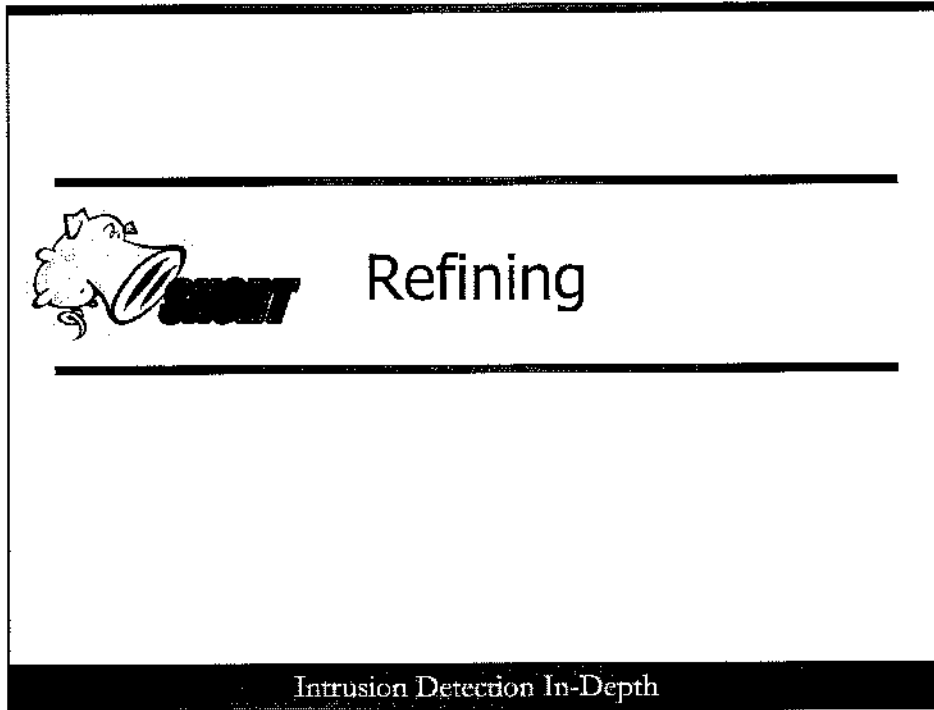
Introduction: Page 3-D

Questions: Approach #1 - Page 5-D
Approach #2 - Page 11-D
Extra Credit - Page 15-D

Answers: Page 16-D

Intrusion Detection In-Depth

This page intentionally left blank.



This page intentionally left blank.

False Negatives and Positives

False Negative	False Positive
<ul style="list-style-type: none">• Snort misses an attack/event• Possible reasons<ul style="list-style-type: none">– Packet loss– No rule written to handle the attack– Badly written rule– Attack obfuscation– Overloaded configuration– Critical path failure	<ul style="list-style-type: none">• Rules go off for non-events• Possible reasons:<ul style="list-style-type: none">– Loosely written rules (too broad)– Unnecessary rules

Intrusion Detection In-Depth

False negatives and positives are the bane of network intrusion analysts. False negatives happen when Snort or any other IDS/IPS misses an event that should have been detected/alerted/logged. There are several general causes of false negatives. One of them is packet loss. Packet loss can be related to hardware or software – or a combination of both. Hardware issues can be caused by network interface cards that cannot keep up with the traffic, an overloaded hardware bus that transports the traffic, insufficient memory and/or processor(s) speed, and inefficient and/or slow speed of writing to disk, or a combination of one or more of these.

Rules or the lack thereof can be the source of false negatives. You cannot detect a zero-day exploit that has no rules. Another rule problem is poorly written rules – perhaps for a specific exploit. Obfuscation is another concern. You could look at encryption as being a type of obfuscation especially if the protocol or well-known protocol is running on a port that usually does not support it. There are other obfuscations such as encodings – base64, unicode, etc.

The configuration of Snort, such as preprocessors selected and values for those preprocessors, may cause Snort to slow down and potentially drop packets. For instance, by default the `http_inspect` preprocessor looks at the first 300 bytes only of the server's response. If you change that to be the maximum packet size of 65535, you lower your risk of a false negative through a failure of inspecting enough of the packet, but you run the risk of slowing Snort down, perhaps to the point where a false negative occurs because Snort is not able to keep up with the packets ingested. As you can see this is a fine balancing act. Performance monitoring is discussed in the Appendix and may be able to help you assess the source of performance problems.

The critical path is the longest execution cycle required to decode, detect and report on the traffic. If this critical path is too slow or is accessed too frequently, packets may be lost because the sensor can't keep up. As much as possible, restrict the sensor to sniffing and analyzing – nothing else like post-processing of alerts.

False positives occur when a meaningless event triggers a rule. What is considered meaningless is unique to each site. A false positive alert can occur if a rule is too broad and erroneously fires on something that is not noteworthy. Another reason is that the rule may be well written, yet not apply to the given site or environment. For instance, suppose there is a stateless rule that pertains to telnet traffic, yet you do not run telnet. An alert that is triggered is essentially meaningless to the analyst because it does not represent a threat in the environment.

Interlude: Writing New Rules

- A new exploit makes its debut; no Snort rules are publicly available yet
- Write a rule to catch the vulnerability in use
- We'll demonstrate a buffer overflow exploit analysis and convert the findings into a Snort rule
- A false negative may arise if you write a rule for a given exploit and not the actual vulnerability

Intrusion Detection In-Depth

Let's combine rule writing with our examination of false negatives. The next several slides demonstrate the methodology for writing and testing a rule associated with a zero-day attack where no Snort rules were publicly available when this vulnerability was discovered.

Keep in mind that it is imperative that you write rules for a given vulnerability and not an exploit. If you write a rule for a particular exploit, it is very possible that there may be a false negative if a different version of the exploit is released. You will see what we mean as we study a vulnerability and first generate a rule for the exploit before we refine it to relate to the vulnerability instead.

We'll follow how a rule is written for a particular IMAP buffer overflow vulnerability.

Interlude: Buffer Overflows

- May allow privilege level of exploited process
- Buffer overflows may allow superuser/root access to a machine
- Overflows can be leveraged to wreak havoc on both internal and external networks
- Example: IMAP AUTH attack

Intrusion Detection In-Depth

Before getting into writing a new rule, it's important to understand how the vulnerability works that you're trying to detect. In the case of buffer overflows, an attacker is usually able to leverage the access gained into root (or superuser) access. Buffer overflows are the scourge of many network server applications, allowing attackers who can exploit them possible widespread access to the other servers and workstations in a network. A successful buffer overflow exploit against a network service that runs with root privileges grants the attacker those same privileges.

So how do buffer overflows work? Generally speaking, poor and misinformed programming is the leading cause of buffer overflow problems today. A buffer overflow works because a programmer fails to properly check or limit how much data is put into a data buffer within a program. These overflows are usually localized to a specific piece of functionality, such as the through the IMAP AUTHENTICATE command in the case of our example.

Interlude: First Describe the Vulnerability

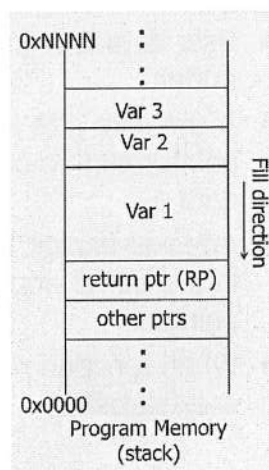
- Inadequate boundary checking within the IMAP server allows attackers to write arbitrary data to the target system and have the target execute the data
- Only effects the AUTHENTICATE IMAP command
- Gives root access to attackers

Intrusion Detection In-Depth

Suppose we learn about a new vulnerability in some IMAP server code. It involves a buffer overflow that when successful, allows the attacker to execute her own code on the target. We've learned that this attack applies to the IMAP AUTHENTICATE command. Most IMAP servers at the time of this given issue ran as root, therefore giving a successful attacker root access.

Interlude: Buffer Overflow Stack Theory

- Programs call their subroutines, allocating memory space for function variables on the stack
- The return pointer (RP) contains the address of the calling function
- Allocated variable space is filled back to front



Intrusion Detection In-Depth

This is a general overview of how buffer overflows work, and may not be 100% correct for a given computer architecture. It does reflect the means that a buffer overflow attack employs to achieve their desired effect.

Buffer overflows work by taking advantage of the way that many modern operating systems work. When a program allocates space for variables in a function, it does so on a structure known as the stack that resides in memory. At the bottom of the stack for a given program's routines is a piece of memory that contains the data for the address where execution of the calling program is to continue when this function call is finished. This piece of memory is known as the return pointer.

The critical point in understanding buffer overflows is that when user data is stored in these allocated variables, the program fills the allocated space "backwards", starting at top end of the buffer and going towards the beginning of that space. Due to the arrangement of variables on the stack, data that overflows the program buffer can end up overwriting the stack's return pointer.

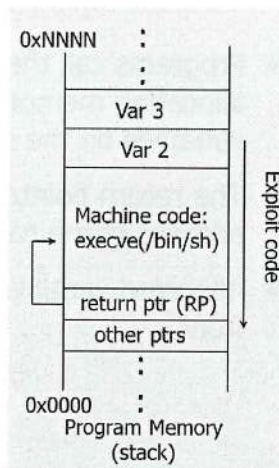
With control of the return pointer the attack code can direct the execution that follows. Ordinarily, the return pointer would point to the calling routine and continue execution at the next statement. But, now, the attacker can point to a location in memory that contains executable code.

For an excellent article on buffer overflows read "Smashing the Stack for Fun and Profit" by aleph1:

<http://insecure.org/stf/smashstack.html>

Interlude: Buffer Overflow Return Pointer Manipulation

- User data is written into the allocated buffer
- If the data size is not checked, return pointer can be overwritten by user data
- Attacker exploit places machine code in the buffer and overwrites the return pointer
- When function returns, attacker's code is executed



Intrusion Detection In-Depth

The logical location in memory to send the return pointer is somewhere that executes the attacker's code. The attacker's code is placed in memory/stack as part of the buffer overflow content that overwrote the stack and the return pointer. The exact location in memory of the attacker's code is often difficult to discern because it is not always predictable. A segmentation fault and crash of the service or process can occur if the attacker's return pointer directs the program execution to a place in memory with a non-executable instruction. This may garner some undesirable attention so the attacker needs to try to avoid this.

A successful buffer overflow and redirection of the return pointer to the attacker's code is potentially dangerous when the attack pertains to an open and listening public port on the host. This means that anyone with access to that vulnerable service and in possession of an effective exploit can potentially gain full control of the target system, possibly using it as entry into the internal network.

Prior to 2004, buffer overflows in server software is the primary means of attack. The trend of attacking client software became more prevalent around 2004.

Interlude: Buffer Overflow Code Example

Susceptible to buffer overflow

```
#include <stdio.h>

int main()
{
    char foo[256];

    strcpy(foo, gets(NULL));

    printf("%s\n", foo);
}
```

Fixed

```
#include <stdio.h>

int main()
{
    char foo[256];

    strncpy(foo, gets(NULL), 255);

    printf("%s\n", foo);
}
```

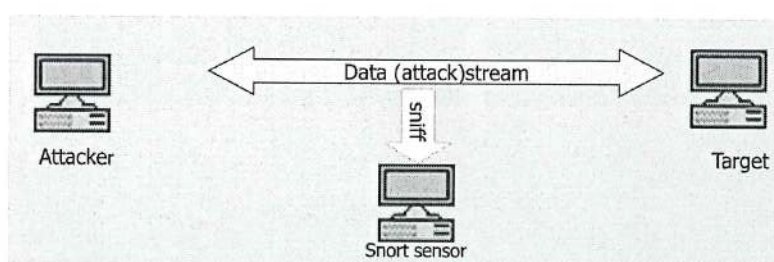
Intrusion Detection In-Depth

Here's a nice little code snippet on the left that illustrates how this problem can happen. We have a piece of code that allocates a 256-byte buffer for user input, then does an unbounded copy of an unbounded amount of data into that buffer. Since the user can input any data she pleases into this buffer, she can essentially take control of the computer if this program is made available to her as a network program with root privileges.

On the right is the same code without the buffer overflow condition included because the string copy (strncpy) limits the size of the copied input to 255 bytes. The strncpy command adds a null terminator character to end of a character array which is why 255 bytes – not 256 bytes – are copied. Note how easy it is to write code that can't be overflowed. Writing secure code is a matter of diligence and good training in proper coding techniques.

Interlude: Generating Packets

- Step One: Get a copy of the exploit
- Step Two: Set up a target host and an attacking host
- Step Three: Set up Snort in packet logger mode `snort -l <logdir>`
- Step Four: Run the exploit under controlled conditions
- Step Five: Analyze and develop signature



Intrusion Detection In-Depth

Now that you are familiar with the theory of a buffer overflow, let's try to write a rule for the IMAP vulnerability. Generating Snort signatures to discover an attack that uses a particular exploit is a fairly straightforward process. The first step is to get a copy of the exploit, if possible. Metasploit is a good place to get exploits; it is found at:

<http://www.metasploit.com>

Once you have a copy of the exploit, you need to set up a vulnerable target host on an isolated test network as the attack target. Once configured, run Snort in packet logger mode (`snort -l <logdir>`). The packet collection can be done with `tcpdump` and written to a `pcap`, if you prefer. After Snort is up and running, execute the exploit code against the target and collect the packets.

Make sure that you perform these activities under controlled conditions, such as in a segregated and monitored lab – not on the production network. When the exploit has completed, you can examine the packets that were collected to try to develop a signature or you can search for a description of the vulnerability and try to compare that with your packet capture and proceed from there.

Interlude: Analyzing Attack Traffic

```

03/05-23:44:12.796225 10.1.1.4:3552 -> 10.1.1.3:143
TCP TTL:64 TOS:0x0 ID:19364 IpLen:20 DgmLen:1118 DF
***AP*** Seq: 0x2B0E465A Ack: 0x66E6C7F3 Win: 0x4470 TcpLen: 20
2A 20 41 55 54 48 45 4E 54 49 43 41 54 45 20 7B * AUTHENTICATE {
31 30 35 33 7D 0D 0A 90 90 90 90 90 90 90 90 90 90 1053}.....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 .....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 .....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 .....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 .....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 .....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 .....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 .....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 .....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 .....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 .....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 .....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 .....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 .....
90 90 90 90 90 EB 38 5E 80 46 01 50 80 46 02 50 .....8^..F.P.F.P
80 46 03 50 80 46 05 50 80 46 06 50 89 F0 83 C0 ..F.P.F.P.F.P....
08 89 46 08 31 C0 88 46 07 89 46 0C B0 0B 89 F3 ..F.1..F..F....
8D 4E 08 8D 56 0C CD 80 31 DB 89 D8 40 CD 80 EB ..N..V...1...@...
C3 FF FF FF 2F 62 69 6E 2F 73 68 9F F2 FF BF 9F ....bin/sh.....

```

Intrusion Detection In-Depth

Here we see the actual exploit packet captured by the Snort logger. A couple of points are worth mentioning about the output captured. First, notice all the 0x90 bytes found in the output. The 0x90 is an Intel x86 architecture NOP – or no operation directive. As the name implies no operation is performed with this code. The reason that you see so many NOP's is because the ultimate goal for the attacker is to be able to place a value in the return pointer that represents the location in the stack where her executable code resides.

Because computing the exact memory location where the executable code resides may not be precisely known ahead of time, if the return pointer accidentally points to a memory location where there is no executable instruction, the program will terminate. Obviously, that is not something that is desirable for the attacker because not only does the attack fail, but the attacker may draw attention to his/her actions. The series of NOP's is often referred to as a NOP sled because it allows the attacker to be less precise in redirecting the pointer. If the pointer falls somewhere in the NOP sled that precedes the exploit code, the series of following NOP's are executable and will eventually lead to the real exploit code.

Another thing that you see above is the "/bin/sh" after the NOP's. This is where the attacker gets a Unix shell and has control over the target host with the privilege level of the vulnerable process.

Interlude: Exploit Characteristics

- Protocol: TCP
- Length: 1118 bytes
- Port number: IMAP, port 143
- Flow: In an established session destined for the server
- Content: "AUTHENTICATE", NOP sled, exploit string (/bin/sh), assembler code

Intrusion Detection In-Depth

There are a number of attributes in the captured packet that can be used to develop a rule to detect such attacks. First the protocol is TCP. Next the packet length is rather large at 1118 bytes. The destination port number is 143, in an established session with the server.

In the payload itself, we saw the content "AUTHENTICATE", the NOP sled, machine code, the /bin/sh, and the assembler code to perform whatever activity the attacker/exploit intends.

Interlude: First Attempt at Writing the Rule

- Combine the characteristics to form the rule
- How could this rule be evaded resulting in a false negative?

Rule header

```
alert tcp any any -> $HOME_NET 143
```

IMAP Port

Rule option

```
(msg: "IMAP Overflow!"; flow:established, to_server; \  
content: "AUTHENTICATE"; content: "/bin/sh"; \  
content: "|9090 9090 9090 9090|"; sid: 50000000;)
```

noop

Intrusion Detection In-Depth

Once the attributes of an attack have been characterized, they can be translated into a rule as above. This rule will be very good at detecting this specific exploit in use, but may not detect variations of it. An attacker may change some aspect of the exploit that will cause the rule to be evaded.

Look at the options we have used in this rule. The content is found in an established session destined for the server. We look for some very specific string matches that may or may not be in every variation of exploit. Suppose that the attack code used a different shell than "sh" – for instance the "/bin/bash" shell instead. This would evade our rule. Also, there are many more assembler operations than the NOP 0x90 that can be used as a "do nothing" instruction that is executable. As an example, an attacker could use the an instruction that represents an operation to subtract 0 from a given value. This is an executable operation that essentially does nothing; it does not change the value of the variable, yet at the same time accomplishes the equivalent of a "no operation".

Interlude: Refined Rule

```

alert tcp $EXTERNAL_NET any -> $HOME_NET 143 \
(msg:"IMAP auth literal overflow attempt";\
flow:established,to_server; content:" AUTH"; nocase; content:"{";\
byte_test:4,>,256,0,string,dec,relative;\
reference:cve,CVE-1999-0005; classtype:misc-attack; sid:1930;\
rev:2;)

```

```

15:57:10.363945 IP (tos 0x0, ttl 3, id 56436, offset 0, flags [DF],
length: 1500) 172.16.10.151.1117 > 172.16.10.200.143: . [tcp sum ok]
1:1461(1460) ack 1 win 5840

```

0x0000	4500 05dc dc74 4000 0306 2828 ac10 0a97	E....t@...((....
0x0010	ac10 0ac8 045d 008f 21d1 9561 c04c 5b60]...!.a.L[`
0x0020	5010 16d0 35e5 0000 2a20 4155 5448 454e	P...5...*.AUTHEN
0x0030	5449 4341 5445 207b 3230 3438 7d0d 0a90	TICATE.{2048}...
0x0040	9090 9090 9090 9090 9090 9090 9090 9090
0x0050	9090 9090 9090 9090 9090 9090 9090 9090

Intrusion Detection In-Depth

Let's be more accurate about the exact rule and base it on the vulnerability – not the exploit. A second revision of the rule improves our first attempt. We see that the content has been changed to be "AUTH" – case insensitive. Perhaps there were some IMAP servers that accepted "AUTH" as a substitute for "AUTHENTICATE". The rule looks for another content of "{" that IMAP uses to denote the number of bytes that follow "AUTHENTICATE". The revision omits the "/bin/sh" so it does not search for specific command shell and it omits NOP codes because the rule is trivially evaded when supplied.

Another option has been added – the byte_test. We will not have a chance to cover byte_test, however you can surmise from the name that it tests one or more bytes for a particular value. This particular byte_test examines four bytes at relative byte 0 after the previous content "{" for a value greater than 256. The value found in this position is a decimal representation of a string. We find an ASCII value of 2048 in that field, which is greater than 256, so the rule should alert. These characteristics have been determined to be representative of the IMAP vulnerability so even if there is a different version of the exploit, these same traits must be present for the exploit to succeed.

Usually you will not undertake writing a rule for a new exploit. However, if you find yourself in a situation where you learn of a new widely used exploit and have no rule, this demonstration can be used to help you approach the process.

False Negatives Solutions

- Packet loss
 - Faster more powerful hardware
 - Refine the critical path to minimize processing not connected to sniffing/analyzing
 - Remove bloated preprocessors, tune the settings on relevant ones
 - Remove rules with no applicable network traffic
- Analysis failures
 - Place sensors in appropriate locations
 - Write rules for a given vulnerability
 - Include all relevant rules
 - Enable all necessary preprocessors
 - Update rules frequently

Intrusion Detection In-Depth

False negatives have several causes. The first category is related to packet loss issues. Packet loss can occur because of slow or underpowered hardware anywhere from the NIC to post-analysis of writing to disk. How can this type of packet loss be avoided? Generally speaking, the answer is to either spend a lot of money or a lot of thought. Putting Snort on faster hardware and with more memory will have positive effects on performance. If you need a less expensive solution to decrease packet loss, consider using BPF to focus on high value traffic only. The use of BPF is supported on the command line via the `-f` switch specifying the location of a file with the filters or the "snort.conf" configuration directive "config bpf_file".

Other packet loss issues are a result of overburdening Snort where packets got bottlenecked and dropped. These are issues like a critical path overload, for instance using several output plug-ins; pare those down so that Snort is predominantly sniffing and analyzing. As we've talked about and will discuss in more detail, delete bloated unnecessary preprocessors that have no relevant traffic. Another preprocessor related problem is using relevant preprocessors but tuning the settings poorly, say for instance by increasing the maximum concurrent sessions that stream5 can process without increasing the memory allocated for TCP processing. It is important to consider the implications of changing a setting because it may have unexpected consequences.

Remove bloated rules, ones with no applicable traffic in the network. You may have multiple sources of packet loss to include any of the ones just discussed, meaning you might have to find and address multiple issues.

The second category of false negatives is some type of analysis failure. A rather obvious one that may not be considered is that you need a sensor in the right place to sniff the traffic you want to analyze. Some sites tend to place the sensors closer to egress aggregation points. That can leave traffic between internal assets uncovered. So, if you need to see traffic from desktop to desktop, you have to place the sensors closer to the first hop aggregation points.

We just examined the consequences of poorly written rules; rules should be accurate and cover a vulnerability – not a given exploit. Both rules and preprocessors may be culprits again in false negatives, but in a different way. The take away is that Snort is not self-aware; it is the job of the analyst to discover the issues.

If you fail to include rules that are applicable for your traffic, you cannot be alerted of noteworthy events associated with missing rules. Once again it is necessary to know the type of traffic that flows across your network. Similarly, you need to include and configure all preprocessors required; otherwise, the related traffic may never be examined and prepared to send to the detection engine.

Finally, you're not going to find malicious traffic associated with newer exploits, if you don't have up to date rules. So, make sure you update your rules frequently for the most current coverage.

False Positive Solutions

- Better vulnerability analysis
 - Tighter rule definition reduces false positives
- Turn off noisy rules that are not particularly informative
 - Just because someone took the time to write a Snort rule and managed to get it into the distribution set does not mean you have to run it!
- Use a SIEM to filter alerts

Intrusion Detection In-Depth

There are a couple ways to deal with rules that generate a lot of false positives. You have to get a better understanding of the vulnerability and rewrite the rule accordingly.

Rules that are just plain noisy and not very informative, ICMP unreachable messages for example, can be turned off altogether. Remember, just because it's in a default Snort rules file does not mean that you are required to run it.

Another solution, although potentially very expensive, is to acquire a Security Information and Event Management (SIEM) tool. These often have methods to filter alerts so that you can examine selected traffic, say for instance DMZ web servers, or high priority alerts easily. This permits you to scrutinize the most meaningful alerts to you or your organization, while filtering out others that may be false positives. This does not get rid of the accretion of false positive alerts, it just eliminates them from your scope.

Sure false positives are not as potentially harmful as false negatives unless they overwhelm the analyst to the point where she misses the true alerts. But, they are definitely annoying so the less, the better.

From Slow to Fast Performance

	Slow	Fast
Logging	ASCII, full	Binary, syslog
Preprocessors	Unnecessary or improperly tuned	Protocols found in the network, tuned settings, especially: <code>frag3</code> : necessary? <code>stream5</code> : ports <code>http_inspect</code> : client/server flow_depth, ports
Rules	Unnecessary, inefficient or improperly written	Protocols found in the network, optimized and precisely written
Sniffing	Slow NIC	Faster hardware and PF_RING for more efficient capture
General	Slow hardware, insufficient memory, processing speed	More powerful hardware or filter for high priority traffic

Intrusion Detection In-Depth

In our discussions, we've highlighted some of the reasons for false positives and negatives. Some of these same items apply to performance improvement. We have recommended the use of default binary logging or efficient syslog logging.

As you know you should use preprocessors that examine traffic found in your network and they should be tuned properly. First disable or comment out `frag3` if there is no fragmentation in your network to avoid searching every packet for signs of fragmentation.

One of the most important preprocessors to tune is the `stream5` configuration for `stream5_tcp` since most network traffic is TCP. `Stream5` assigns many ports for client and server TCP traffic that is to be reassembled. This means that the streams are tracked and packet payload is combined, consuming resources. Look at these settings and compare the port values with ports used on your network. You can run `perfmon`, discussed in the Appendix, to generate statistics about ports in use. Remember, removing ports from the `stream5` lists means that the individual packets are still examined, but not reassembled.

`Perfmon` can assist you in tuning `stream5` parameters `memcap` and `max_tcp` settings. These values may not appear in the preprocessor settings in "snort.conf" if the defaults are used. Too many concurrent TCP sessions defined in `max_tcp`, but not enough memory defined in `memcap` can cause performance issues.

There are some `http_inspect_server` settings for the `http_inspect` preprocessor parameters that should be examined and tuned. HTTP accounts for a high percentage of traffic on most networks so any efficiency that can be gained, should be addressed. There are parameters for `client_flow_depth` and `server_flow_depth`. The default value for each is 0, meaning that the entire payload is examined. The default value used to be 300 bytes

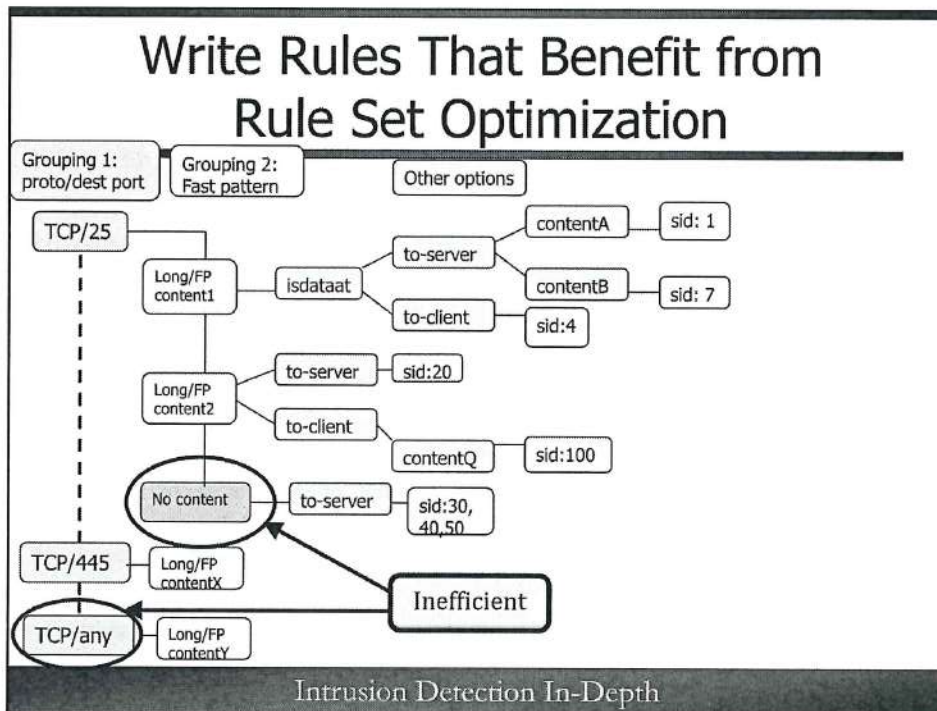
since it was believed that most of the malicious activity would be discovered before that byte count in both client and server HTTP payload. It is possible that this condition is no longer true as attackers develop new methods such as embedded javascript that may be found deeper in the packet/stream. It's a hard call to make whether or not to alter these values, however if you have severe general performance degradation and a high percentage of HTTP traffic, this might be worth attempting to tune.

Another parameter associated with the `http_inspect_server` that can be tuned is the list of ports used to transport HTTP. Snort is able to perform protocol decode initially based on well-known ports and not the payload itself. Examine this long port list and tune it for your network by eliminating those ports that don't apply and adding any that do apply.

You know that you need to be careful with the rules that you include as well as exclude. We'll discuss how to make rules more efficient on the next slide.

Being able to keep up with the throughput on the network is essential since that is where the entire process starts. You'll need a NIC card that is able to sniff at the required speeds and you may consider installing `PF_RING` for more efficient capture.

Finally, hardware can be a limitation in general – from the bus speed, employing multi-bus architecture – separate ones for transferring traffic from the interface code to memory and memory to disk, to the amount of memory, processor speed and even writing to disk. Obviously, this is fixed with better hardware. And, if your budget does not permit this, consider BPF use to focus on high priority traffic only.



Before you can write efficient rules, you have to understand how Snort optimizes the rule set after Snort is started. There can be many rules in a rule set and matching each packet against every rule would be extremely inefficient. To optimize the process, the rules are grouped first by protocol and destination port. For instance, in our example, all rules with TCP port 25 are grouped, then TCP 445, and then TCP with any destination port and so on for every rule with a different protocol and port combination.

So this means as a packet arrives, Snort quickly determines which set of rules is applicable to the packet by protocol and port as a first match. If there are rules with protocol/"any", all packets/streams with that protocol must be further examined by rules in the "any" path. This is extremely inefficient. If all rules have assigned destination port values, any packet that does not match those values is not evaluated. So, it is dropped early from analysis. This is why "any" destination port assignments are expensive.

Next, the rules are grouped according to a fast-pattern match designation or the longest content. The pattern matcher is optimized to evaluate a packet that has fallen into the given a protocol/port path and quickly examine it for any of the long or fast_pattern content found in this grouping of rules. If there a rule has no content match to include pre rules with no anchor, and negative content, any packet with that protocol/port pairing must be evaluated. Again, this is inefficient because this is another earlier analysis spot to eliminate packets quickly.

Finally, rules within the same protocol/port/long content are grouped by shared matching options, such as flow to server or flow to client, etc. Eventually, Snort groups rules with all matching characteristics. Now, a packet that has fallen into the port/protocol/content path can be compared to the rule option values or conditions that must exist. Early mismatches (bail conditions) such as flow direction, or number of bytes present in payload, eliminate packets more quickly. If the packet/stream has all the necessary attributes, Snort knows which rule(s) to apply to it.

So you see the importance of having a destination port, content, and early bail conditions, on a rule when possible.

Snort Performance: Writing Good Rules

- Avoid the use of any for the destination port
- Use longest content match, or most unique shorter content match via `fast_pattern`
- Use "fast_pattern" keyword, if appropriate
- Avoid rules with no content match, negative content match, or pcre with no content match anchor
- Place "bail" conditions early in the rule

`flow, flowbits, isdataat`

Intrusion Detection In-Depth

To sum up the previous slide. Whenever possible supply a destination port since this is the earliest efficiency that can be realized. The "any" destination port requires every packet with a given protocol to be examined.

Whenever possible, take advantage of Snort's fast pattern matching algorithm by supplying the rule with the longest possible content value that you expect to find. This doesn't necessarily need to apply to an exploit/vulnerability or malicious part of the packet or stream; it just must be found in the packet or stream. The fast pattern matcher selects the first, longest, non-negative content from each rule. If a rule that you write has more than one content, yet has some unique shorter content that is sought, use the `fast_pattern` keyword modifier to inform the pattern matcher to use it instead. Use a content anchor with pcre rules to get the benefits from the pattern matcher. And, avoid complex pcre expressions since they can be computationally expensive.

Snort processes rule options in the same order that you supply them. So, use stream state options such as **flow**, **flowbits**, and **isdataat** immediately so that the rule engine can bail early and not waste CPU cycles on packets that don't match. The **flowbits** option permits rules to track state where one rule defines a condition that must be found before a second **flowbits** rule triggers predicated upon a given setting in the associated **flowbits** rule. The **isdataat** option examines if data exists at a particular byte placement, often used to discover buffer overflows.

You see how rules that have no destination port, or no content or negative content create a "bucket" of rules causing all packets that have arrived at that bucket path to be analyzed. It's analogous to the programming statement like a "switch" or "case" that lists all possible values for a given variable and typically has a catch-all default statement in the event that none of the values matches. The packet may eventually end up in the catch-all category requiring it to go through wasteful checks.



Updating

See Appendix of Snort Material for Updating material

Intrusion Detection In-Depth

This page intentionally left blank.

Writing a Snort Rule for a CVS Exploit

Workbook

Exercise: "Writing a Snort Rule for a CVS Exploit"

Introduction: Page 25-D

Questions: Page 26-D

Answers: Page 31-D

Intrusion Detection In-Depth

This page intentionally left blank.

Snort Good Reading

- Snort User's Manual
- USAGE File
- Various README files
- Man page
- Snort-users, snort-sigs, snort-devel, and Emerging-sigs mailing lists

Intrusion Detection In-Depth

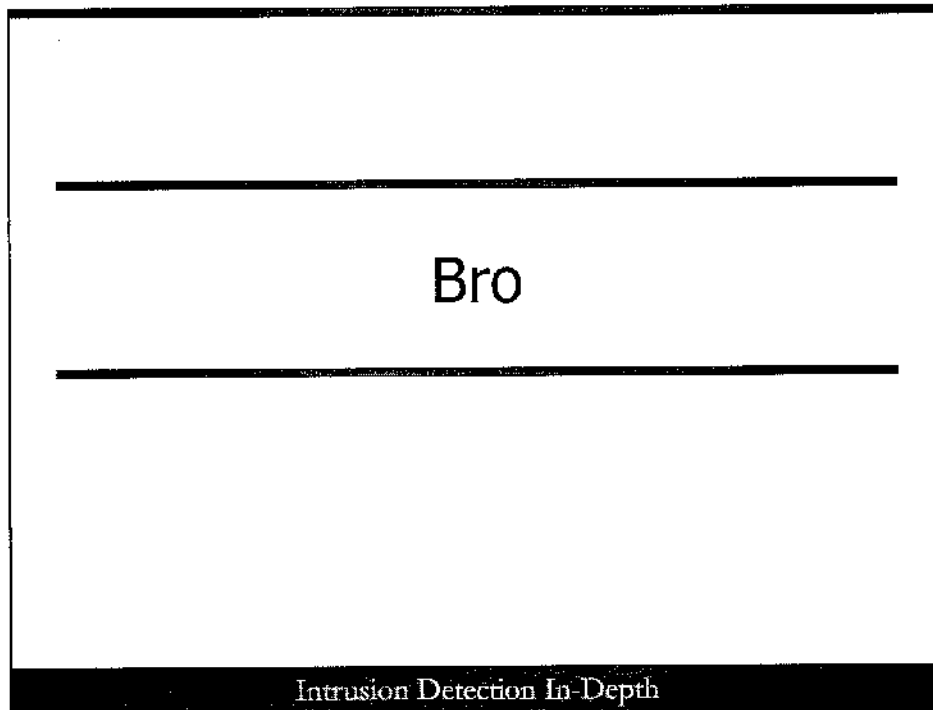
As we've mentioned, there is a variety of documentation that you should take a look at to familiarize yourself with the system. A good comprehensive reference guide is the Snort User's Manual found at <http://manual.snort.org>.

Look in the "snort/doc" directory; there are several files that are useful, including README.* files that describe different protocols and preprocessors, a USAGE file for Snort and the Snort manual in PDF format. The README.* files are written by the developers. This has the benefit of being a direct source for the information as opposed to translated by someone into the manual. It has the disadvantage of sometimes being less coherent.

Additionally, for people who have questions and want the quickest and most accurate answer, sign up for the Snort-users mailing list at <https://www.snort.org/community>.

There are a few other lists available, Snort Sigs (Snort rules), and Snort Developers (Snort development). Check out the "mailing lists" page at <https://www.snort.org/community> for more information.

Also, look at Emerging Threats for additional mailing lists: <http://lists.emergingthreats.net/mailman/listinfo/>.



Perhaps you've heard of Bro or perhaps not. Though it has been around since the mid 1990's, in the past it has been deployed predominantly at academic institutions. As such it may have gotten an undeserved reputation. When a very savvy friend was asked why not give Bro a test drive, she lamented that it would be too hard to learn because it was from an academic environment. That's true; it was and it is still developed with academic support.

Yet, anyone can use Bro. It used to be much more difficult to install, but that is no longer true today. And, in fact, there is an open source distribution Security Onion that is available with both Snort and Bro as well as many other network traffic-centric tools. Much like any software, there are levels of expertise that allow you to customize it to use more complex features. As with Snort, you can install it and do basic configurations to identify the protected network and use the default features. After a while, you may find that you want to alter the way it performs, requiring a deeper understanding. The depth of the knowledge required is commensurate with the difficulty of the customization you want to support.

Bro is considered to be "extensible", an industry buzz word that means that it can easily accommodate customization and expansion. There are stub entry points in Bro called events where the user can place customized code. In other words, the software anticipates the desire for customization and makes provisions to interface with the user's code. Optional use of these features does require some advanced knowledge, but many novice users may not wish to employ these.

What Is Bro?

- Open source software originally written by Vern Paxson
- Intended to run on commodity hardware
- Bro cluster configuration supports growth in network coverage
- Network traffic analysis framework
- Generates log files of network activity
- Event-driven
- Default functionality to assist with analysis and detection
- Customizable for site-specific analysis using scripting language
- Supports rudimentary signature creation

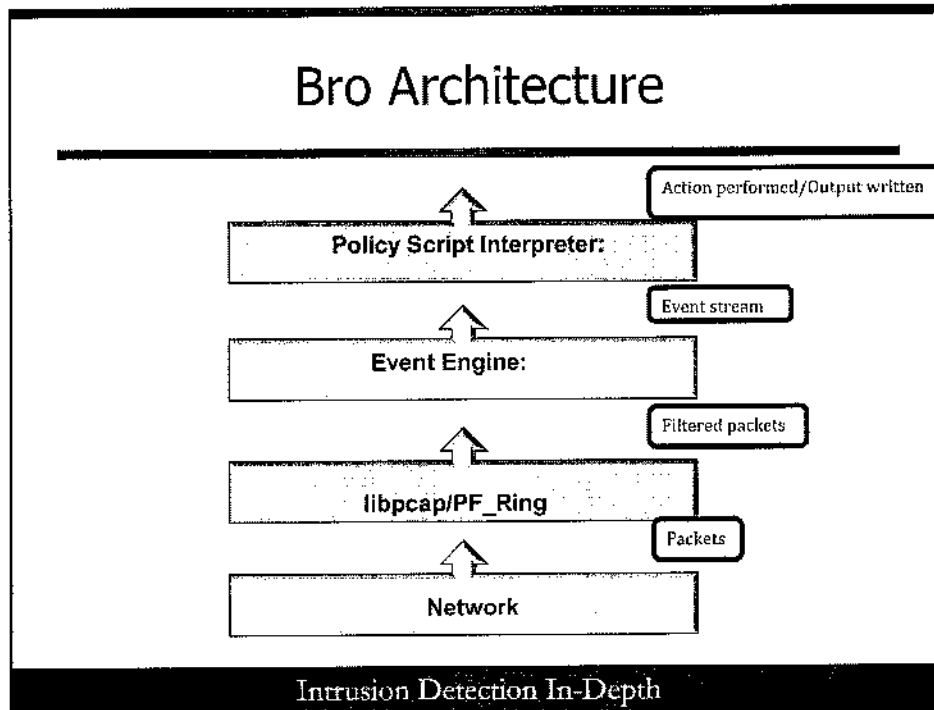
Intrusion Detection In-Depth

Bro is open source software originally written by Vern Paxson at Lawrence Berkeley Labs. Today, Bro is supported by several different developers at the International Computer Science Institute in Berkeley and the National Center for Supercomputing Applications in Urbana-Champaign, Illinois. Bro even has a commercial incarnation known as Broala. Bro is deployed in just about every industry, but seems to have the most ardent following at academic sites.

Bro has been associated with academia since its inception and as such supports the concept of open source free software on affordable commodity hardware. Bro can be configured to run on a single host for smaller networks or in a cluster for larger networks. The cluster configuration facilitates expanding network coverage – perhaps when network traffic or speed increases – with the addition of more hosts that sniff traffic. This doesn't require an upgrade or port to more powerful hardware, it simply relies on adding more Bro "workers" to the existing Bro infrastructure.

Bro is fundamentally different than a signature-based IDS/IPS. It provides traffic analysis on every connection and records notable details of each in various tab separated logs for post processing. Other IDS/IPS solutions focus and report on detection of anomalous or malicious traffic. Bro is capable of doing this, but this is not its exclusive purpose. It also logs what it defines as events and records summarized data about each connection. Events can be normal or anomalous network events, perhaps something as simple as a new network connection. These events are pre-defined in the software and can be customized by the user to do some post-event processing that includes creating some kind of notification or action of noteworthy activity. On Day 5 you'll learn about the concept of alert versus data-driven sensors. Bro falls in the category of data-driven since it collects the data, but it is up to the analyst/software to determine its significance.

Bro has its own complex and full-featured scripting language that permits site customization for just about any detection and analysis. This, of course, requires someone to be proficient in using the language. The scripting language is extremely powerful and has functionality equivalent to many other high level languages like Python, for instance. However, the Bro scripting language's purpose is to facilitate parsing and analysis of network traffic. Other languages like C or Python support analysis of network traffic by importing libraries for this purpose. Bro, first and foremost analyzes traffic and has support for other processing secondarily. Bro does support rudimentary signature creation, yet relies more on its scripting language for more advanced analysis.



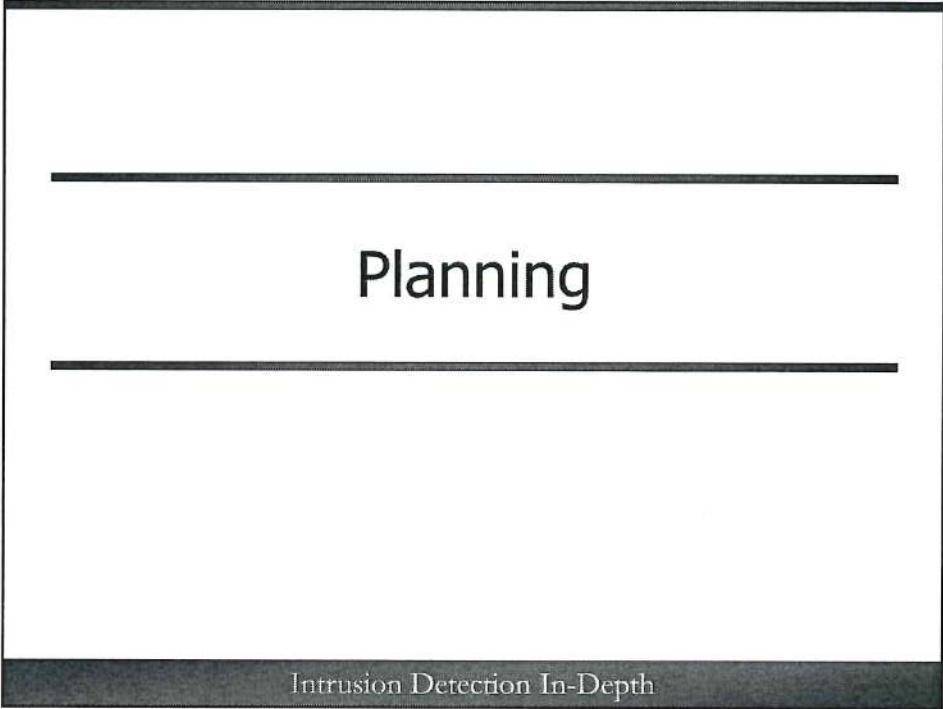
Bro's architecture relies on libpcap at its base to sniff and filter packets much like Snort. PF_RING can be wrapped around libpcap when there are multiple Bro sniffing processes on a single host, permitting the traffic to be directed to individual cores on multi-core architecture hosts. Think of it as host load balancing.

The event engine takes the traffic reassembled as streams and generates network events for several hundred milestone or noteworthy states. These are normal states such as a TCP connection established, or an observed HTTP request. These pertain to different network layers, application protocols, along with various stages and associated data of processing those protocols.

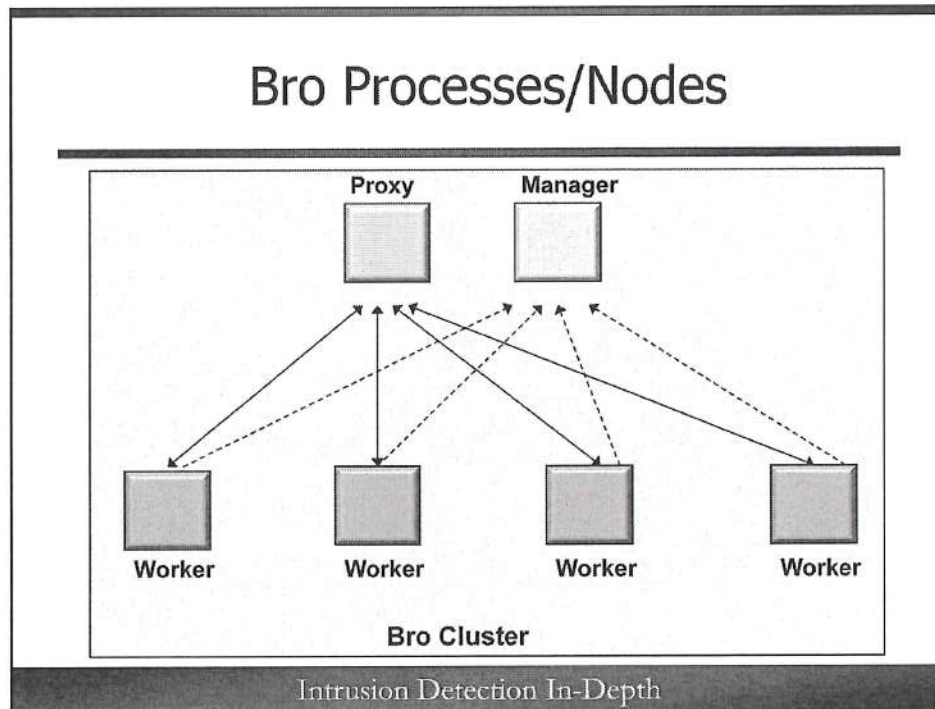
The event engine first examines traffic using a protocol analyzer that is associated with a well-known port. However, it also has a process known as Dynamic Protocol Detection (DPD) that interprets the traffic not based on the port, but on the content of the payload. If someone were to try to tunnel HTTP traffic over an unusual port, Bro would still be able to interpret it as HTTP using the DPD.

The policy script interpreter uses Bro's scripting language to interpret Bro's defined events. The events provide entry for user-written scripts to add logic to the event, describe the event, and possibly create actions such as generating notices, sending e-mail, or recording the event in text files. For instance, say you would like to be notified of a known attacker IP address connecting to your web server. There is a Bro event for each connection. This is the event you would use to invoke your customized script – check the connection for both the IP addresses of the attacker and the web server, as well as the TCP destination port of 80 or 443.

You can "raise a notice" to cause a message of your choice to be written to a file called "notice.log". Alternatively, you could "raise a notice" with an action of e-mailing a chosen recipient(s). The term "raise a notice" means creating some kind of notice for the event to signal that the analyst should examine the activity. It is similar to a Snort alert.



This page intentionally left blank.



Bro has three different functional nodes – the worker, the manager, and the proxy. Together these three nodes form what is known as a Bro cluster. Bro even has its own communications protocol to transmit Bro traffic between nodes.

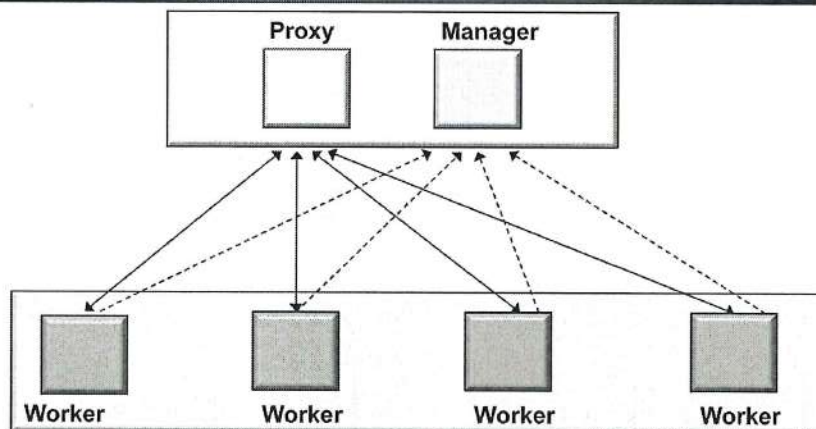
The worker, as the name suggests, is the workhorse that sniffs the traffic, reassembles it into streams, and performs protocol analysis on the streams. The worker is commonly known as a sensor for most other IDS/IPS solutions. The deployment form that the worker takes can be varied. A given worker can be an individual separate host, or a process on the same host as other workers sharing one or more cores on that host. A worker can share the same host as the other two nodes – the manager and the proxy. This is a standalone implementation. The number and placement of the workers depends on the traffic volume and the type of hardware available for traffic collection. Since the worker sniffs and processes traffic, it needs to be deployed with fast and abundant memory, and CPU speed that is more likely found on several multi-core hosts.

The manager is the central log and notice collector, creating a single log for all the traffic collected by the workers. These log messages can be processed into actions such as e-mail. Communications between the manager and workers are initiated by the workers.

The proxy is responsible for making it appear as if all the workers are operating in a single unified environment instead of as different hosts or different processes on different cores on the same host. It synchronizes the state of Bro by sharing information about active hosts and services on the network as well. There may be multiple proxies in a Bro implementation if one is insufficient. The proxy and manager may be run on the same physical host if the number of implemented workers and associated activity are manageable.

A logical distinction is made between standalone mode where all nodes are on the same host or cluster mode where at least one node is a remote host. However, standalone mode is actually a cluster just on the same host.

Bro Cluster on One Physical Host: Standalone Mode

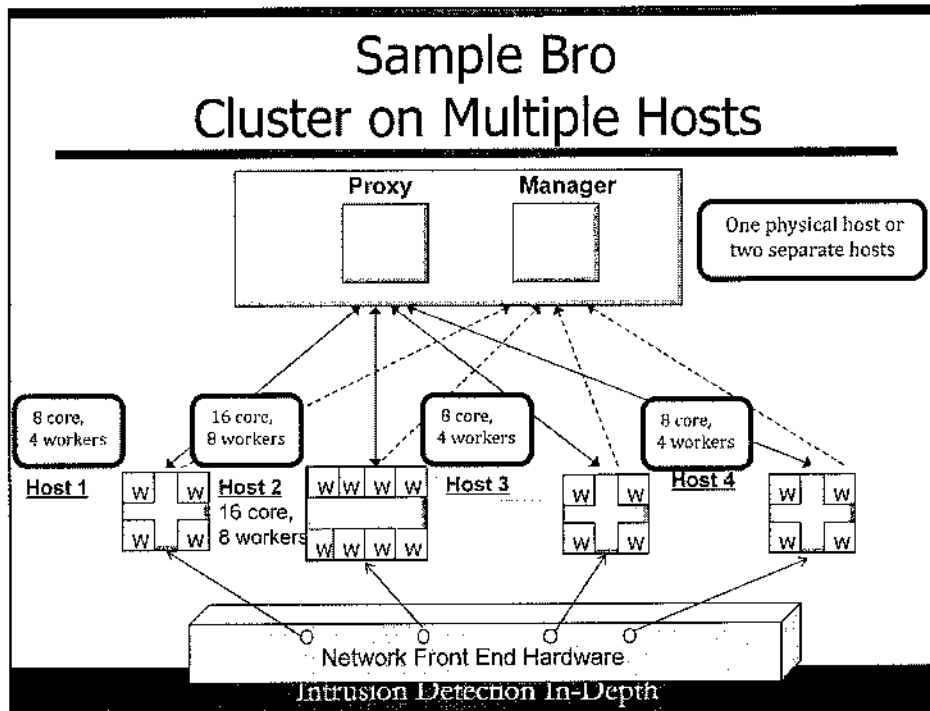


Intrusion Detection In-Depth

Bro is very flexible in its configuration options, facilitating capacity planning and scaling. The simplest, though least flexible option, is to put a Bro cluster on one physical host more commonly known as a standalone implementation. There is a single proxy, a single manager, and one or more workers that are manifested as distinct processes on the host.

The workers process the traffic collected from one or more network interfaces. The use of the open source software PF_RING on the host balances the sniffed traffic so that it is split among the workers. Traffic that belongs to the same stream is sent to the same worker. In other words, a TCP session is not split among workers. And, Bro for the most part is stream oriented, not packet oriented like Snort. One exception to this is the per packet processing performed when signatures are used.

Standalone mode makes managing the Bro cluster easier and probably cheaper than having multiple physical hosts, although a problem arises when the traffic and processing exceed the capacity of the host. One solution in this case is to buy a more powerful host and reinstall and customize Bro on it. Another solution is to filter the traffic captured and analyzed using BPF so that only high value traffic is processed. A final solution is to create a distributed cluster with one or more nodes on different hosts as you'll see on the next slide.



Bro is able to easily manage growth when it is run in a dynamic enterprise environment. The standalone Bro implementation requires new hardware and reinstallation and customization of Bro when capacity is exceeded. Running workers on different hosts permits the addition of more hardware and a simple reconfiguration on the manager to identify the new worker(s) that were added to accommodate the increase in traffic volume or speed.

The slide shows a Bro cluster consisting of the proxy and manager residing on one physical host and four other physical hosts running worker nodes. Each physical host may be multi-core, able to run multiple worker nodes. The number of worker nodes per host is limited by the number of cores on the host. A rule of thumb, although very dependent on the nature of the traffic examined, is that the number of worker nodes should be between one half to three fourths the number of cores. For instance, Host 1 has 8 core processors and 4 workers. Hosts 3 and 4 are identical and Host 2 has 16 core processors and is running 8 workers. You see that the number of workers never exceeds 50% of the number of cores.

This configuration necessitates the inclusion of some kind of front end hardware that acts as a load balancer to split the network traffic among the physical hosts. The hosts still employ PF_RING to act as a single host based balancer to distribute traffic among the workers residing on the host.

When you plan your Bro installation requirements, it is best to overallocate the number of physical hosts, host memory, and processor speed for the workers. It is very difficult to make recommendations of the precise number or amount of resources you will need because that is dependent on many different factors including the traffic protocols that require analysis, the speed of the link, the configuration of the individual packets – smaller packets are more plentiful and require more resources for reassembly, and where the workers are placed in the network. Some approximate metrics are given on the following slide.

Rough Metrics

- Metrics per core
 - Each core can support $\sim .5 - .75$ workers per
 - For instance - 12 cores should have maximum 6-8 workers
 - Each Bro node (manager, proxy, worker) starts a child process, too many processes overwhelm the processor
 - Each core can sniff ~ 80 megabits per second
 - Each core requires $\sim 2-4$ gigabytes of memory

Intrusion Detection In-Depth

Seth Hall, one of Bro's primary developers, is reluctant to give metrics – understandably since some users take the figures as the absolute unwavering truth. As you will learn on Day 5, a sensor's capability to handle packets is based on far more than the product deployed. Other factors include the hardware platform, the NIC card, as well as the mixture of traffic flow, to name a few. Maximum throughput rates tend to be greater for simple shorter packets such as DNS versus HTTP packets. If the mixture of traffic is high on DNS and low on HTTP, you may get very different numbers than a site that has the reverse situation.

The measurements cited are per CPU core for Bro 2.3.x code – subject to change with new Bro releases. Each core can support about $.5 - .75$ workers. This means that if you have 12 cores, you can support approximately 6-8 workers. Each Bro node forks off a child process that consumes and potentially overwhelms resources if there are not enough cores per node. The child process is responsible for the communications between nodes. This has the role on the worker node of separating the communication processes from sniffing the traffic, which should be the worker's main job.

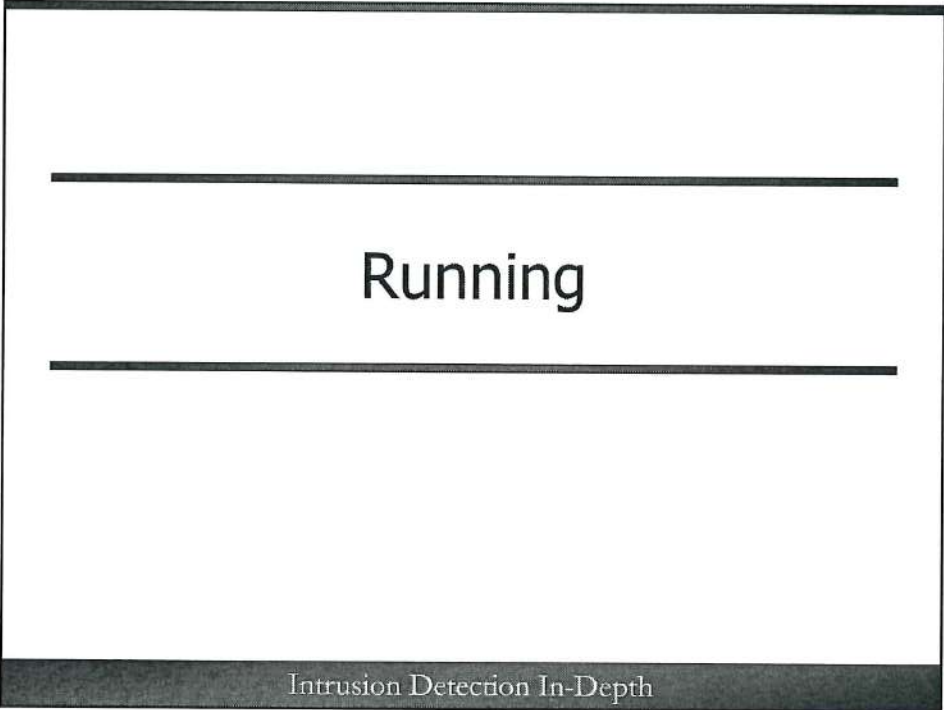
Each core requires roughly 2-4 gigabytes of memory. Each core can sniff approximately 200 megabits per second. Seth also recommends a minimum of 8 cores to get the benefits of a cluster that has more than a handful of workers.

Installation/Configuration

See Appendix of Bro Material for
Installation/Configuration material

Intrusion Detection In-Depth

This page intentionally left blank.



This page intentionally left blank.

Broctl

- BroControl shell to start, stop, update, check on status, output of configuration of running Bro
 - Can be run in interactive mode
 - Can be run on command line
- Broctl reads PREFIX/etc/node.cfg file to run in assigned mode:
 - Standalone: manager, proxy, and worker(s) all on one host
 - Cluster: manager or proxy or worker(s) on different host(s)

Intrusion Detection In-Depth

We've mentioned BroControl, invoked via the "broctl" command, that is used to start, stop, install, update, check on the status, run the affiliated cron, and more. It is possible to run "broctl" in an interactive mode or on the command line.

BroControl is run in a shell type of environment when used interactively. Commands are entered by the user such as "status" command below. This shows a cluster configuration.

```
broctl
[BroControl] > status
```

Name	Type	Host	Status	Pid	Peers	Started
manager	manager	192.168.43.1	running	20832	1	30 Aug 05:23:00
proxy-1	proxy	192.168.43.1	running	20868	1	30 Aug 05:23:02
worker-1	worker	192.168.43.19	running	23893	2	30 Aug 05:43:03

BroControl can be run from the command line and supplied an argument, such as the status command:

```
"broctl status"
```

This mode is more likely if you have some kind of automated invocation of the process like we saw with the cron entry.

Bro node.cfg Configuration for Standalone Mode

PREFIX/etc/node.cfg

```
[bro]
type=standalone
host=localhost
interface=eth0
```

Intrusion Detection In-Depth

The above default configuration, found in "PREFIX/etc/node.cfg" causes Bro to run in standalone mode. You need only assign the proper interface name used for sniffing on the Bro host.

Starting Bro in Standalone Mode

```
bro@hostname: sudo broctl
Welcome to BroControl 1.1
Type "help" for help.
[BroControl] > install
removing old policies in /usr/local/bro/spool/installed-scripts-do-not-touch/site ... done.
removing old policies in /usr/local/bro/spool/installed-scripts-do-not-touch/auto ... done
creating policy directories ... done.
installing site policies ... done.
generating standalone-layout.bro ... done.
generating local-networks.bro ... done.
generating broctl-config.bro ... done.
updating nodes ... done.
[BroControl] > start
starting bro ...
[BroControl] > status
```

Name	Type	Host	Status	Pid	Peers	Started
bro	standalone	localhost	running	23335	0	30 Aug 05:40:43

Intrusion Detection In-Depth

Let's fire up Bro in standalone mode. The "broctl" command must be run with elevated privileges so we use sudo to invoke the command as root. You will see a welcome message and the guidance for help. The "install" BroControl command must be run to load Bro's policies either when first starting it up or any time a change is made to some configuration, script, or signature.

Once this is successful, you need to "start" Bro; a "status" command logically enough displays the status. We see Bro is running in standalone mode and has no peers (remote nodes).

Bro node.cfg Configuration for Cluster Mode

```
PREFIX/etc/node.cfg
## Example clustered configuration. If you use this, remove the [bro] node above.
[manager]
type=manager
host=192.168.43.1
#
[proxy-1]
type=proxy
host=192.168.43.1
#
[worker-1]
type=worker
host=192.168.43.131
interface=eth0
```

Intrusion Detection In-Depth

Alternatively, Bro can be run in cluster mode. When configured so in the "PREFIX/etc/node.cfg" file, the default configuration statements to run in standalone mode must be commented out:

```
#[bro]
#type=standalone
#host=localhost
#interface=eth0
```

Next, you must uncomment out the statements to define the manager, at least one proxy, and at least one worker. You can supply any name for any of these nodes in the [node] name – such as [myworkerbee] as long as you assign the proper type (manager, proxy, or worker) for the node. The manager and the proxy do not have sniffing interfaces, only the worker(s) does and must have the interface defined. The proxy, like the worker(s) can reside on a remote host too, although in the example the manager and proxy reside on the host with IP address 192.168.43.1 while the worker resides on the host with IP address 192.168.43.131.

Starting Bro in Cluster Mode

```
bro@hostname: broctl
[BroControl] > install
....
generating cluster-layout.bro ... done.
generating local-networks.bro ... done.
generating broctl-config.bro ... done.
updating nodes ... done.
[BroControl] > start
starting manager
starting proxy-1
starting worker-1 ...
[BroControl] > status
```

Name	Type	Host	Status	Pid	Peers	Started
manager	manager	192.168.43.1	running	20832	1	30 Aug 05:23:00
proxy-1	proxy	192.168.43.1	running	20868	1	30 Aug 05:23:02
worker-1	worker	192.168.43.131	running	23893	2	30 Aug 05:43:03

Intrusion Detection In-Depth

Starting Bro in cluster mode is different because we invoke "broctl" as user "bro". Remember that communication is required among all nodes via SSH. We've configured a "bro" user to communicate over SSH using SSH keys, requiring no password or passphrase entry.

Like standalone mode, use the "install" and "start" BroControl commands. The "install" command not only loads all the scripts and configuration files for the manager, it also copies over the requisite Bro software and support files to the remote nodes(s). This time you see all the different cluster nodes starting. The BroControl "status" command shows the status of each of the nodes. The "peers" designation defines the number of remote peers – for both the manager and proxy-1 on the same host, this represents the one worker peer. The worker node has two peers since both the manager and proxy, though residing on the same host, are considered two individual nodes.

Bro Output

- Several logs are generated by Bro as default output
- Tab delimited columns of values that can be fed into Bro "cutting" or post-processing of your choice
- Each log reflects different activity depending on the traffic observed and frameworks employed
- Correlation of traffic from same session/stream is possible with the shared UID field
- Default location PREFIX/logs

Intrusion Detection In-Depth

Logs are the default output for Bro reporting. Values are tab delimited into associated field columns. The logs generated depend upon the activity that is observed as well as customization that you might use such as the Notice framework or signatures, for instance that we will examine in the next section. Any output from a particular connection is assigned a common Unique ID (UID) that is logged in every file where this connection activity appears. This makes it easier to correlate activity for a given connection by searching for that UID in all of the logs.

Bro places all of its log files/directories in "PREFIX/logs". There is a directory named yyyy-mm-dd (with the appropriate year, month and day values) that houses all the log files per day. By default, Bro rotates logs on a daily basis, although this value is configurable in "PREFIX/etc/broctl.cfg". The current day's activity is found in "PREFIX/logs/current" although it is really stored in "PREFIX/spool/bro" in standalone mode. There is a symbolic link from "PREFIX/logs/current" that points to "PREFIX/spool/bro".

Bro generates new log files every hour, assigning each a file name that reflects the type of log followed by a time range indicating when log recording began and stopped. Normally, this is the beginning and end of the hour. Though, if you happen to stop and then restart Bro, two different sets of logs are created. The first one reflects the set of current logs that are saved with a timestamp range of when Bro was started, usually the beginning of the hour, and an end time stamp of when Bro was stopped. A new set of logs is created with a name that will later contain a time range of the Bro restart time and the time reflecting when the the hour was up or when Bro was stopped before the hour was up. Bro compresses each of these hourly files at the end of the day.

Sample of the Types of Logs Generated

Log Name	Purpose
conn.log	Initial IP/protocol connections
conn-summary.log	Statistics/summarizes activity
known_hosts.log	New hosts seen in past hour
known_services.log	New services seen in past hour
dpd.log	Dynamic protocol detection
weird.log	Anomalous activity
loaded_scripts.log	Scripts loaded upon start/restart
reporter.log	Severity of issues with Bro
software.log	Determines version numbers of already detected protocols
Various protocols (http, dns, ssl, smtp,etc.)	Logs with relevant activity pertaining to a given protocol

Intrusion Detection In-Depth

Logs are numerous; and you should become familiar with the type of content some have along with the potential for use in host or network forensics. We'll discuss many of these in more detail in upcoming slides. The "PREFIX" reference in a file name refers to the install directory.

The "**conn.log**" creates an entry for the initial connection of TCP and UDP traffic and summarizes some characteristics such as the number of packets and bytes generated by either side of the connection.

The "**conn-summary.log**" provides a very useful summary of connection activity for the duration of time that the log spans, usually an hour.

The "**known_hosts.log**" records every complete TCP handshake, and keeps track of IP addresses in use on a network each day.

The "**known_services.log**" records an IP and port which responded to a SYN, and protocols detected in the session are also logged per day.

The "**dpd.log**" records protocols discovered on non-standard ports by using Bro's Dynamic Protocol Detector (DPD) that works by using "heuristic methods". See "PREFIX/share/bro/base/frameworks/dpd/dpd.sig" for all the protocols available for detection.

The "**weird.log**" is a catch-all log for anomalous behavior. This might be something like an evasion attempt of Bro, malformed connections or traffic that is not protocol compliant.

The "**loaded_scripts.log**" lists the Bro scripts that were loaded upon start/restart from the supported subdirectories of "PREFIX/share/bro" – "base", "policy", and "site".

The "**software.log**" attempts to determine software versions of already identified protocols.

The "**reporter.log**" reports on Bro status errors. There are several ratings "informational" that as the name implies is for information, "message" that signals a potential problem, "warning" that indicates a non-fatal, but definite problem where Bro doesn't terminate, and "error" that is a fatal error that terminates Bro.

There are various protocol-specific logs for protocols that Bro detects, such as "http.log", "dns.log", etc.

Customization

Intrusion Detection In-Depth

This page intentionally left blank.

Customizing Bro: Policy Neutral to Site Specific

- One of the ultimate goals when developing Bro was to make it policy neutral
- Several means of allowing customization to site-specific policy:
 - **Scripts:** post-event processing allows you to develop code in Bro's robust scripting language
 - **Signatures:** allows you to create rudimentary rules to find noteworthy activity
 - **Notice capability:** post-event processing allows you to associate a notion of importance to events and perform some kind of action

Intrusion Detection In-Depth

A primary philosophy that the developers of Bro felt was fundamental is what they deem site neutral policy. This permits a site to customize evaluation of any given event to determine its significance and what action to take as a result of that evaluation. Snort, on the other hand, has a one-size-fits-all philosophy of ascribing a priority to each rule/alert, thereby assigning a universal value reflecting the potential harm of the detected activity. This notion of priority may be very different depending on acceptable activity per site.

There are three basic ways to customize Bro's detection and post-detection activity. The first way is by using the Bro scripting language. One description of this language by one of Bro's developers, Robin Sommer, in his presentation "The Open Source Bro IDS Overview and Recent Developments" found at <http://www.icir.org/robin/slides/Bro-CACR-Indianapolis.pdf> is "A Python for network traffic analysis". It permits the user to develop basic or very advanced post-processing after one of Bro's events is triggered. The Bro scripting language was developed expressly for the purpose of analyzing network traffic as opposed to some more general language like Python that needs to retrofit libraries or routines to facilitate network traffic analysis. This language enables a given site to create code for what is considered noteworthy traffic on that particular network. This is the primary method Bro uses to expose the detection process to the user.

The second way to customize policy and detection is via signatures. Bro signatures are very basic methods of identifying and recording or reporting on some pertinent characteristics about the traffic, mostly based on content, and modifiable by such characteristics as traffic IP addresses and ports. You can specify the location of the content – say in the entire payload – or some more protocol-specific location like the HTTP header. Regular expressions are supported to describe the content that is sought.

The final way to customize Bro for your site is the Notice Framework to "raise a notice", somewhat like a Snort alert. As you know, the policy neutral philosophy of Bro means that Bro does not assign a judgment of good or bad to events or activity detected. Essentially, it detaches the discovery of activity with an assignment of importance. Therefore, each site can assign what it considers to be a judgment of the nature of the activity.

For instance, suppose that some outbound IRC traffic is discovered on a particular network. If this is a university environment, chances are that this is perfectly acceptable and considered unimportant. Yet, if that same activity is observed on some kind of government network, it may be considered a policy violation that requires immediate action. The follow-up post-detection activity can be customized by each site.

There is extensive and sophisticated support for the policy neutral philosophy. In fact, at times it may appear to be overkill for your needs and may require you to understand more than you care to know. The Notice Framework plays a big part in implementing this philosophy as you will see in some upcoming slides. Just understand that while you may think it is esoteric and unnecessarily complex, the ultimate reason is to make Bro more customizable for every site that deploys it. It is not necessary to use all facets of the Notice Framework, if you use it at all.

Bro was largely developed at academic institutes where the assumption is that most future Bro users are fairly savvy about coding. Therefore, some of the concepts, constructs, and implementations may seem arcane. But, if you can push through the initial learning curve fog, Bro has amazing potential. Also, it is always possible to use default configurations or make some basic changes without ever having to know about scripting, signatures, or notices.

How Would We Find cmd.exe Traffic?

- Using Bro's scripting would be complex and require a lot of skill
 - No supplied event to trigger script
 - No script
- Using Bro's signatures would be much easier, relying on:
 - Assignment of appropriate IP addresses
 - Assignment of ports and TCP protocol
 - Use of a regular expression to find the malicious payload

Intrusion Detection In-Depth

Imagine that you wanted Bro to find the cmd.exe traffic that we examined in the section on Snort. There are really two options available to us; the first is to use Bro's scripting language; the second is to use Bro's signature framework. A script would allow the scrutiny of the text to be as precise as you want given all the features of the language. There must be both an event trigger to cause the script to fire and an existing script that performs the search that we want, making it a simple and manageable process for a Bro novice. Unfortunately, none exists so we need to rely on Bro's signature capabilities.

We cannot be as precise as we'd like with a signature, yet we can define appropriate IP addresses, ports, protocols (TCP, UDP, ICMP, IPv6, and ICMPv6), and payload to find. Our best bet is to use Bro's support of regular expressions to find the content in the payload. We'll examine how to build a signature in the upcoming slides.

Customization Option: Bro Signatures

- Simple rules to define:
 - Conditions to match
 - Action to perform upon match
- Different types of conditions:
 - Header: Examine specific header fields/values
 - Content: Look for particular payload
- Two different types of action:
 - Events
 - Trigger a Bro script
 - Enable protocol analyzer
- Learn more:
<http://www.bro.org/sphinx-git/frameworks/signatures.html>

Intrusion Detection In-Depth

As you know, Bro supports the notion of signatures. Snort signatures provide much more functionality than Bro signatures because Snort signatures are considered to be the primary way that noteworthy activity may be found. Since Bro has a scripting language that is capable of performing many of the more advanced features of Snort signatures, its signature support is lightweight in comparison. So do not expect the same level of functionality in Bro signatures.

Bro signatures define conditions to match in network traffic and the action to be performed upon match. Bro defines several types of conditions that can be used in signatures; we'll look at the two most commonly used. The first is known as a "header" condition that is capable of examining source and destination IP addresses, a list of IPs, either designated as a single IP address or in CIDR format. As well, it can match on source and destination ports or a port list. And finally, it can match on three protocols TCP, UDP, and ICMP.

A second type is a "content" signature that can include any of the fields available in the header signature, in addition, match on content. The content can be anywhere in the payload or in some defined fields as we will see.

There are two possible actions upon signature match. The first is to raise a signature event in the "signatures.logs" and "notice.log" file containing pertinent data about the packet/stream that caused the signature to match. You will see that we can actually trigger a Bro script to be invoked from a signature.

Another action, the "enable" action tells Bro to use a specific dynamic protocol detector. Typically, traffic is analyzed as the protocol associated with a well-known port. Port 80 is analyzed by Bro's HTTP protocol detector, as port 25 is analyzed using Bro's SMTP protocol detector. Bro has the capability to identify that the traffic is not protocol compliant. The appropriate protocol analyzer is applied if Bro can match the analyzed protocol in use with one it is able to understand. Say for instance that someone was using DNS TCP port 53 as an HTTP tunnel. It is possible for Bro to understand that the traffic is HTTP and parse it accordingly. An "enable" action identifies the protocol parser for Bro to use, otherwise Bro attempts to figure it out, but may not do so correctly.

In our example we thought someone was running intermittent HTTP traffic on a port not normally associated with HTTP – DNS TCP port 53. You could create a signature that examined traffic on TCP port 53 and had content normally associated with HTTP – say "GET" or "User-Agent:" to confirm your suspicions by enabling your signature with an action of "enable http".

A more comprehensive description of signatures can be found in the Bro signature documentation found at: <http://www.bro.org/sphinx-git/frameworks/signatures.html>.

Header Signature

<header field> <comparison operation> <value list>

- Header field
 - Source/destination IP address (single, list, CIDR notation)
 - Source/destination port (single, list)
 - IP protocol: TCP, UDP, ICMP
 - Supports the use of BPF-like designation for these fields
header <proto>[<offset>:<size>] [& <integer>]
- Comparison operator
 - ==, !=, <, <=, >, >=
- Value list
 - Single or multiple values

Intrusion Detection In-Depth

We'll examine two different types of Bro signatures "header" and "content". When using the header type of signature, you can match three header fields: IP addresses, ports, and three IP protocols limited to TCP, UDP, and ICMP. Source and destination IP addresses can be expressed as a single IP address, a list of IP addresses, or in CIDR notation. Source and destination ports allow a designation of a single port or list of ports. Finally you can examine traffic for any of the three IP protocols available.

Bro supports the use of BPF-like syntax to select a subset of bytes in the header fields by identifying the offset into the appropriate protocol and the number of bytes to examine – like tcpdump – either 1, 2, or 4. As well, you can perform bit masking if you need to.

The comparison fields are relatively straightforward – equal, not equal, less than, less than or equal, greater than, or greater than or equal.

And finally a comparison value must be supplied – either a single or multiple values.

Simple Header Signature

```
signature proto-port{  
  ip-proto == tcp  
  src-port == 30333  
  event "Protocol/Port Test"  
}
```

```
jnovak@judy:~/bro-run/protoport$ bro -r cmdexe.pcap -s proto-port.sig
```

Output placed in signatures.log



Intrusion Detection In-Depth

proto-port.sig
cmdexe.pcap

This is a simple header signature assigned a name "proto-port" in the signature definition. It simply looks for the IP protocol of TCP and a source port of 30333 as we saw in our cmd.exe traffic. The signature format is shown above where the word "signature" identifies this as a signature, followed by the signature name in this example "proto-port". Then the actual signature is placed in between left and right braces. The "event" parameter of a signature defines the message associated with the signature, much like the Snort "msg".

Whenever you execute Bro in readback or command line mode – as opposed to live mode – be aware that Bro creates log files from the output that are placed in your current working directory. It is best to create a new directory or use an existing one to contain the log files to isolate them from files used for other purposes.

We can perform a simple run to test the signature by invoking Bro to read cmdexe.pcap using the "-r" parameter followed by the pcap name, with the "-s" command line switch specifying the file name containing the signature – in this case a file we named "proto-port.sig".

Once again many logs are created, but for our purposes, we want to look at the "signatures.log" that is created when a signature is triggered. The following slide has the contents of that file.

Contents of signatures.log

```
#fields  ts          uid          src_addr    src_port    dst_addr
#types   time          string       addr        port        addr
1410983624.505015 CgbFvZOi2aG63NSo7 192.168.11.24 30333      184.168.221.63
```

```
#fields  dst_port    note          sig_id
#types   port        enum          string
48938   Signatures::Sensitive_Signature proto-port
```

```
#fields  event_msg          sub_msg    sig_count  host_count
#types   string             string     count      count
192.168.11.24: Protocol/Port Test (empty)   -          -
```

Intrusion Detection In-Depth

The signatures logs is displayed in an altered format for easier reading. There is too much output to fit on a single line, consequently there is a lot of confusing wrapping. Normal output has the field names, associated variable types, and values on consecutive single wrapped lines. For coherence, related fields, types, and values are shown together.

Much like the other log files the "signatures.log" file has a timestamp for the activity, source and destination addresses and ports. The field "sig_id" is the name of the matching signature, the "event_msg" is a descriptive message of the signature matching event, and finally the "sub_msg" is defined as "extracted payload data or extra message" – there was no such message for our signature as you see it says (empty).

It is interesting that the "signatures.log" and the signature parameter names themselves have different field names for the same fields found in many of the other logs. For instance "src_addr" represents the source IP address in the "signatures.log", but is known as "id_orig.h" in the others. One of the reasons that this particular field is mentioned, as previously stated, Bro documentation makes a point of calling the source of the traffic the "originator" and the destination the "recipient", perhaps for clarity to the developers. Yet, here is the "signatures.log" that uses the more common identification of source and destination.

Header Signature With CIDR IP List and List

```
signature source-ip{
  ip-proto == tcp
  src-ip == 192.168.0.0/16, 10.0.0.0/8
  src-port=30333
  event "Source IP Test"
}
jnovak@judy:~/bro-run/source-ip$ bro -r cmdexe.pcap -s source-ip.sig
jnovak@judy:~/bro-run/source-ip$ cat signatures.log | bro-cut -d src_addr src_port
dst_addr dst_port event_msg
192.168.11.23 30333 192.168.11.24 48938 192.168.11.24: Source IP Test
```



This signature is somewhat more sophisticated using a list (more than one value) of possible destination IP addresses denoted in CIDR format. The source IP address can be anything with a 192.168.x.x or 10.x.x.x value. You can also specify port lists in much the same way using commas as delimiters. If all of the signature conditions are met, an event message "Source IP Test" should be found in the "signatures.log" file.

We haven't used the "bro-cut" command before. It is a way to extract values from a log without listing all the headers that include field names and types. You can dump all the fields by doing a Unix "cat" and piping the output to "bro-cut". The -d option formats the output in a humanly readable form. You can also output specific fields only by supplying the names to the "bro-cut" command. This means you have to know the field names; they can be found at the top of the "signatures.log" file as we saw on the previous slide.

We run Bro reading the "cmdexe.pcap", using the -s switch to inform Bro that the signature file name "source-ip.sig" contains a signature. The resulting "signatures.log" file is similar to the one we just inspected except the event message contains this signature's event "Source IP Test".

Header and Content Signature Seeking Simple Payload Match

```
signature simple-content{
  ip-proto == tcp
  src-ip = 192.168.0.0/16
  src-port == 30333
  payload /Microsoft Windows/
  event "Microsoft Windows found in payload"
}
```

```
jnovak@judy:~/bro-run/simple-content$ bro -r cmdexe.pcap -s simple-content.sig
```

sig_id	event_msg
simple-content	192.168.11.24: Microsoft Windows found in payload



Like Snort, Bro has the capability to look for some content in the payload by using a "content" signature. You can supply the name of supported different parts of the payload that can restrict the search fields. There is a generic "payload" designator that examines up to 1K bytes of payload, the limit imposed by Bro for performance efficiency. This can be altered if so desired using the `dpd_buffer_size` value to redefine it as we will later see.

There are other protocol-specific fields to search such as the "http-request-header". It searches for the content in the HTTP request header only. There is also "http" to search the entire HTTP payload, "http-request-body" for searches in the HTTP request body portion, "http-reply-header" for the reply HTTP header, and "http-reply-body" for the body of the HTTP reply. Finally, you can use "ftp", and "finger"; "finger" is a legacy protocol that is not commonly used today, probably a vestige from Bro's early days.

This signature file "simple-content.sig" simply looks for the existence of "Microsoft Windows" in the payload. The content is contained within the two forward slashes much like the syntax for regular expressions. The content can be expressed as a regular expression as we'll see.

How to Express Signature Payload Using a Regular Expression

Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

Original text

content : "Microsoft Windows"; depth:18; content:"Copyright |28|c|29| 2009";
distance:0; content:"Microsoft Corporation"; distance:0

Snort rule

Wildcard

Wildcard

Microsoft Windows * Copyright (c) 2009 * Microsoft Corporation

Beginning of line
followed by "Microsoft
Windows"

Followed by
"Copyright (c)
2009"

Followed by
"Microsoft
Corporation"

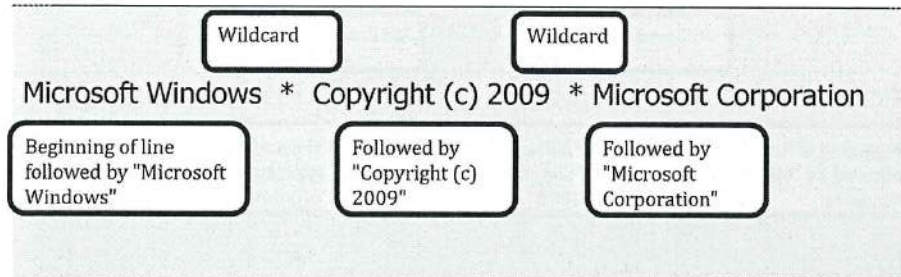
Intrusion Detection In-Depth

Like Snort, Bro supports the use of regular expressions when searching payload content. Let's make our signature more accurate by trying to duplicate, somewhat, the Snort rule criteria using a regular expression.

At the top you see the actual text of displayed terminal output when you start cmd.exe. The Snort rule is more precise than we'll make the Bro signature using simpler terms in our regular expression. We want to search for "Microsoft Windows" found at the beginning of the payload, followed by an unspecified number of bytes/content before searching for "Copyright (c) 2009", followed by an unspecified number of bytes/content before searching for "Microsoft Corporation".

Signature with Regular Expression Payload Match

```
signature cmd-exe {  
  ip-proto == tcp  
  src-ip = 192.168.0.0/16  
  src-port == 30333  
  payload /^Microsoft Windows.*Copyright \(c\) 2009.*Microsoft Corporation/  
  event "Windows 7 cmd.exe remote execution"  
}
```



Intrusion Detection In-Depth

Here is our regular expression supplied as the signature payload. In this context the "^" sign means that the content is found at the beginning. We follow that with the text "Microsoft Windows" and then a wildcard designation of ".*" to account for the unknown or unpredictable content. Next we supply the "Copyright (c) 2009" string.

While we used hex to express the parentheses in Snort, we need the escape character backslash before the parentheses to designate that we are looking for the string value. Otherwise there is confusion with the syntax because regular expressions use parentheses to group parts of it together. We finish the regular expression with another wildcard and the content of "Microsoft Corporation".

The Snort rule language has the capability to easily add efficiency by restricting the depth of part of the content. It is possible to make this rule more efficient using some additional features of regular expression syntax, however that is beyond the scope of focus.

Header and Content Signature Seeking Payload Match

```
signature cmd-exe {  
  ip-proto == tcp  
  src-port == 30333  
  payload /^Microsoft Windows.*Copyright \(\c\) 2009.*Microsoft Corporation/  
  event "Windows 7 cmd.exe remote execution"  
}
```

```
jnovak@judy:~/bro-run/cmdexe$ bro -r cmdexe.pcap -s cmdexe.sig
```

```
cat signatures.log | bro-cut sig_id event_msg
```

```
sig_id          event_msg  
cmd-exe 192.168.11.24: Windows 7 cmd.exe remote execution
```



We place the signature in a file named "cmdexe.sig" and run it through Bro. The contents in "signatures.log" reflect that it worked as expected.

Procedure for Loading Your Signatures for Live Traffic Capture

- Place your signature file in PREFIX/share/bro/site directory or subdirectory you create
- Edit PREFIX/share/bro/site/local.bro
 - Add a @load-sigs line with the name of your signature file name
- Tell Bro to reload in broctl:

```
[BroControl] > install
```

```
[BroControl] > restart
```

- Generate some traffic to trigger the signature
- Go to directory PREFIX/logs/current -> PREFIX/spool/bro
cat signatures.file | bro-cut

Intrusion Detection In-Depth

The examples shown using signatures were invoked while reading traffic from a pcap file. The signature file was identified with the command line -s switch. What if you wanted to load these same or different signatures when running Bro via "broctl" to read live traffic? The procedure to load your custom signatures is very similar to loading your custom scripts.

First, place all of your signatures in either "PREFIX/share/bro/site" or a subdirectory beneath it that you create to keep your personal signatures.

Next edit "PREFIX/share/bro/site/local.bro" to add the name of your signature file. The "@load-sigs" is used to specify your signature file name. You have to inform Bro to reload its startup files by invoking "install" and "restart" in broctl.

Now, generate traffic to trigger the signature and look for output of success in the file "signatures.log" in the "current" log directory by examining it with bro-cut. As you can see the "current" log directory has a symbolic link with the spool directory.

Customization Option: Bro Scripting

- Full-featured language with many of the same capabilities as higher level languages such as C, Python, etc.
- Offers network-related representations of traffic – hosts, ports, protocols (HTTP, SMTP, etc.), protocol fields/states – `http_request`, `new_connection`
- Permits you to write your own code or customize existing code for site-specific purposes
- Functionality like Snort capability to write preprocessor
- Complete understanding of this complex language is beyond the scope of the class, we'll just touch on the basics
- Default scripts found in `PREFIX/share/bro/base/`
- Optional policy scripts found in `PREFIX/share/bro/policy`
- Site-specific scripts should be placed in `PREFIX/share/bro/site`

Intrusion Detection In-Depth

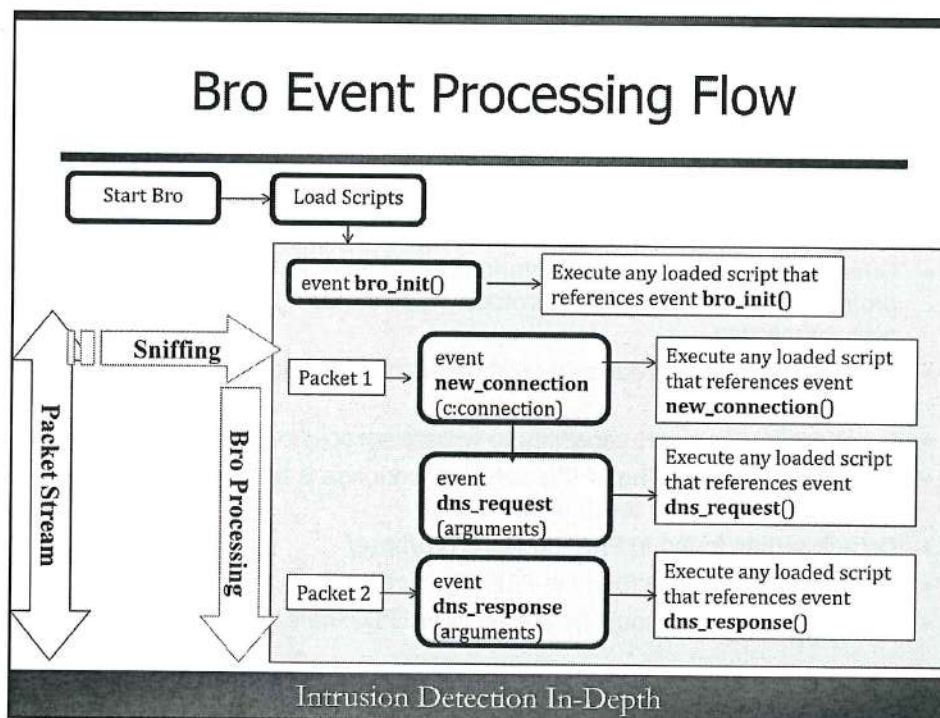
One of the most useful and advanced features of Bro is its own scripting language that was predicated upon interpretation of network traffic rather than using a library or retrofit as done by a more generic language like C. This means that you are likely to find far more functionality and native support for network protocols that you would like to process.

The language offers the analyst a platform to customize code. There are many default scripts that are included in Bro for different protocols and different processing. You may find that what is included is complete enough for your purposes. However, you may either want to embellish what currently exists or add completely new functionality for site-specific processing. Bro scripting requires knowledge of programming constructs and may have a steep learning curve depending on your familiarity and comfort with programming.

We will cover some rudimentary concepts associated with scripting and create some simple scripts. But, this is by no means comprehensive. An entire day, if not more, could be devoted to teaching the Bro scripting language. You may not be interested in learning Bro scripting because of its complexity or because you don't require this degree of customization. Many users never learn to write customized code for Snort, yet are able to masterfully use Snort. It is useful to know the scripting language is available and the material that follows will assist you in creating and executing basic scripts.

The Bro documentation available, especially on scripting, is scattered and makes assumptions that you are aware of the environment in which scripts are run. It does not take you step by step demonstrating how to start and how to proceed as the following material intends to do.

The scripts that come with Bro are installed in "`PREFIX/share/bro/base`" with a file extension of ".bro." These scripts are not intended to be changed; if the user wants to make customizations, those scripts should be placed in the directory "`PREFIX/share/bro/site`". The directory "`PREFIX/share/bro/policy`" contains scripts that are considered to be less universally required and may be more computationally involved; therefore the user must elect to load them. It should be mentioned that the site-specific scripts need to be identified to Bro; this is done in the configuration file "`PREFIX/share/bro/site/local.bro`". We'll learn how to do this later.



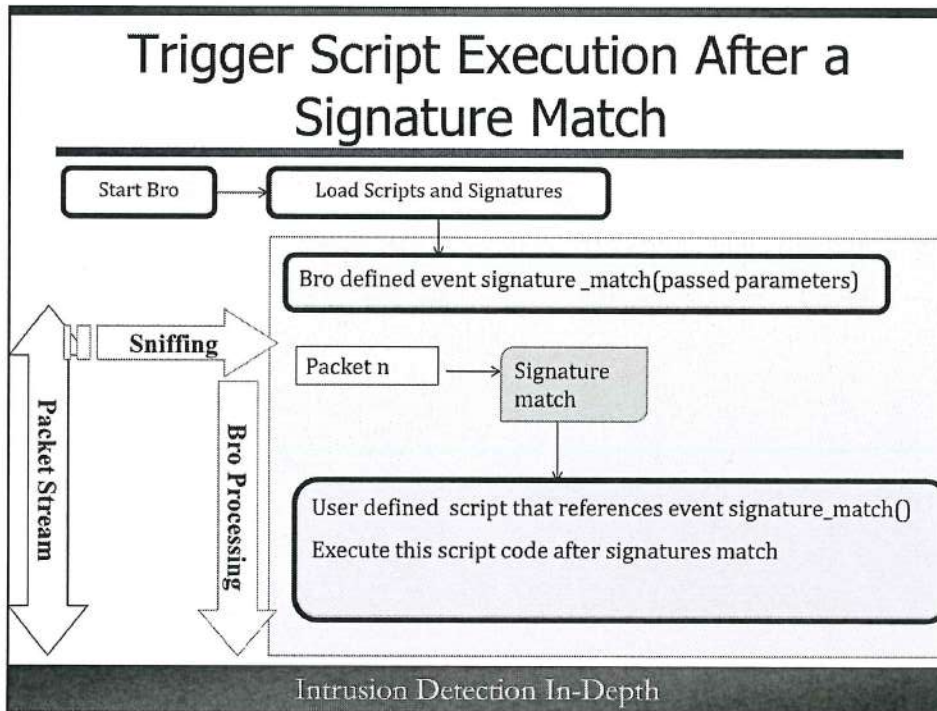
The notion of a Bro event is very powerful since it presents an opportunity for customization by adding your own scripts. If you decide that you'd like some custom processing, you need to understand that a Bro event is your vehicle for identifying the trigger condition to execute your code. Let's scrutinize the process in more depth.

Upon starting, Bro examines the "\$PREFIX/share/bro/site/local.bro" file for entries that begin with "@load filename". The "@load" is equivalent to a Snort configuration file or C programming language "include" statement in that it references specific files, in this case Bro scripts, that should be loaded. The file names can be those included with Bro or those that you create. These files/scripts reference a particular Bro event found in "PREFIX/share/bro/base/bif/event.bif.bro", "PREFIX/share/bro/base/bif/plugins/Bro*bif.bro" or "PREFIX/share/bro/base/protocols/protocolname/main.bro" along with the code that is to be executed upon Bro's processing of that event.

For instance, in our example, Bro is started and one of the events that Bro encounters in its processing is the "bro_init" event. The event processes any loaded script(s) that references the "bro_init" event. If for example, you wanted Bro to write a message that Bro started, perhaps with a date and time, you would supply a script that references "event bro_init" that accomplishes this. You would place the location of that script in the "local.bro" file using "@load filename" and it would be loaded after installed and restarted. Now, after Bro starts and determines that the conditions are met for "bro_init" event, your script is triggered and executed.

Each packet sniffed thereafter may cause Bro to follow a different code path perhaps based on the protocol. Suppose packet 1 is a DNS request and packet 2 is a DNS response. Bro will decode the packets and determine that they contain the DNS protocol. Say this is a new never-before seen connection. If there is native Bro code or a script that you added to be processed upon seeing a new connection, it will be processed when the event "new_connection" condition is encountered. Eventually, Bro will follow the code path for processing DNS. All events that are associated with DNS can trigger an associated, perhaps user written, script to be executed.

What is the takeaway here? First, a Bro defined event is the entry point for adding custom code. Second, not all Bro-defined events will be processed for every packet. An event is triggered when event-specific conditions exist, like a new connection is detected or when a particular protocol is encountered. Bro-defined DNS event conditions are met only when DNS is detected and other specifications are met – like a DNS request or response.



Indeed, the notion of events and associated scripts can be arcane so let's try to give the discussion some gentler context. We are going to execute a Bro script that we write that will be triggered when a specific signature finds a match. As mentioned, Bro loads some default scripts and signatures upon being invoked. One of these scripts is defined as an event named "signature_match()" that is automatically triggered when a signature match occurs. By default, there is no accompanying user-defined script that is triggered by this match or event. Bro has its own scripts to perform signature matching processing such as logging data to the "signatures.log" file.

But, suppose you wanted additional processing to occur when a particular signature found a match in packet n above. We would have to write our own code for the script that began with the line "event signature_match(passed parameters)" where "passed parameters" are the names and variable types passed; we'll look at those in an upcoming slide. Bro would execute our code after any signature match. Therefore, we have to examine the unique signature identification string (the name we assigned to our signature) that is passed to our code before performing some signature-specific activity.

Let's Examine the Format of Our Script

Keyword
"event"

Selected
event name

Passed
parameters

```

event signature_match(state: signature_state, msg: string,
data: string)
{
  Code begins here
  if (state$sig_id == "cmd-exe")
  {
    print "Process cmd.exe script code";
  }
  Code ends here
}

```

Code begins here

If statement to match
signature name; {} used
to specify the code block

Code followed by end of
line character ";"

Code ends here

Intrusion Detection In-Depth

Let's look at the code and format of our script to process upon signature match. As you are aware, you need to identify an existing Bro event that will cause your script to trigger – in this case "signature_match". This is the event that occurs after a signature matches. It causes Bro to look for any code it knows about that references the "signature_match" event to be executed.

This is accomplished by supplying the keyword "event" followed by the actual name of the event you selected to trigger your script. Any arguments that are to be passed to your script are enclosed between the parentheses. The "signature_match" event has three defined passed parameters named "state", "msg", and "data" that are Bro variable types of "signature_state", "string", and "string" respectively.

Bro uses the left and right brace characters to isolate a block of code – in this example, the beginning and end of the code associated with the event processing. Other examples of related code are those executed after conditional and loop statements. The "print" statement format is much like those used in other languages. The semi-colon is used to identify the end of a line of code.

We check for the unique signature identification, the name we gave to our signature, of "cmd-exe" before we execute the code. We want this particular code to apply only to that particular signature since this event is called whenever any signatures matches. How did we know the variable name "state\$sig_id" to use to identify the signature identification field? We'll discuss this on the next slide.

Obviously, there are many more features and capabilities to the language than described here. Documentation for Bro scripting can be found at:

<http://www.bro.org/sphinx-git/scripting/index.html#writing-bro-scripts>

Another option for learning syntax and scripting is to examine the scripts that come with Bro.

State Data Structure

```
print fmt("Structure of parameter state ===> => %s ", state);
```

Entire state
data structure

state\$*sig_id*

```
Structure of parameter state ===> [sig_id=cnd-exe, conn=[id=[orig_h=184.168.221.63, orig_p=48938/tcp, /tcp], orig=[size=0, state=4, num_pkts=2, num_bytes_ip=0, flow_label=0], resp=[size=141, state=4, num_el=0], start_time=1410983624.504544, duration=0.09251, service={^}^], addr=[st=0, history=ShAd, uid=uninitialized], op=uninitialized, conn=uninitialized, extract_orig=F, extract_resp=F, dhcp=uninitialized, dns_state=uninitialized, ftp=uninitialized, ftp_data_reuse=F, ssl=uninitialized, http=uninitialized, irc=uninitialized, modbus=uninitialized, smtp=uninitialized, smtp_state=uninitialized, syslog=uninitialized], is_orig=F, payload_size=141]
```

Intrusion Detection In-Depth

It turns out the variable named "state" is a nested data structure in Bro's parlance. A nested data structure allows embedded data substructures. Let's dump the variable "state" using a formatted print statement. This new print statement replaces the one used in the script in the previous slide to be executed upon signature match.

Bro denotes each data structure with a name followed by "=", followed by all the variables in that structure enclosed in "[" to begin and "]" to end. For instance, you see the "conn", "orig", and "resp" data sub structures (highlighted with arrows under the name) all contained within the "state" signature data structure. In order to identify a given variable, you must begin at the base structure, in this case, "state" and identify any other nested structures. The "sig_id" field is denoted as "state\$*sig_id*"; the "\$" serves to dereference or delimit each of the nested data structures as well as any variable in the final nested data structure.

As you will see, the variables and data structures names associated with a given protocol are often a mystery unless you look at Bro's associated event file.

Another option is to print the containing data structure, like "state", as we did to reveal the names of the data structures and variables.

Run Our Script

```
user@user:~/bro-run$ more sig-event.bro
```

```
event signature_match(state: signature_state, msg: string, data: string)
```

```
{  
  if (state$sig_id == "cmd-exe")  
  {  
    print "Process cmd.exe script code";  
  }  
}
```

```
jnovak@judy:~/bro-run/sig-event$ bro -r cmdexe.pcap -s cmdexe.sig sig-event.bro
```

```
Process cmd.exe script code
```

```
jnovak@judy:~/bro-run/sig-event$ ls *.log
```

```
conn.log notice.log signatures.log
```



Intrusion Detection In-Depth

cmdexe.sig sig-event.bro
cmdexe.pcap

We discussed the practicality of creation and execution of Bro in a separate directory when running signatures in readback mode. The same advice applies to running scripts as you see in the bottom panel of the slide that the environment uses the bro-run/sig-event directory for this purpose.

We run Bro using the "cmdexe.pcap" as usual, include the signature "cmdexe.sig" as we did before, but also make Bro aware of our new script named "sig-event.bro". As you see the message "Process.cmd.exe script code" appears, showing that the script was triggered by the signature event match. Some of the logs that were produced are the same as those generated when running a signature with no script.

This is a demonstration of how to trigger a script from a matching signature. The code executed is not especially productive since teaching the intricacies of Bro's scripting language is not the intent.

Simple Script Without Signature Framework

- More often than not, scripts are not triggered from a signature match
- While not especially useful, let's create an example script that prints "Bro started" upon Bro invocation
- This will familiarize you with the methodology

Intrusion Detection In-Depth

Scripts that users write are typically associated with a triggering event other than the signature match. It was easier to start your understanding of scripts using the signature as a trigger. Now let's examine how you would trigger a script you write from a non-signature event.

Let's start with a simple example. The only thing our amazing script will do is print "Bro started" upon starting Bro. Remember that any script that you write needs to be associated with a Bro event. As will be discussed on the next slide, there are several hundred events and we just need to find the appropriate one.

What Are the Names of the Available Events?

- Events define trigger conditions and entry point for your customized scripts
- Found in:
 - PREFIX/share/bro/base/bif/event.bif.bro
 - PREFIX/share/bro/base/bif/plugins/Bro*bif.bro
 - PREFIX/share/bro/base/protocols/protocolname/main.bro

bro_init:	connection_SYN_packet:
mobile_ipv6_message:	teredo_packet:
tcp_contents:	protocol_violation:
udp_request:	icmp_error_message:
login_failure:	ftp_request:
smtp_data:	mime_content_hash:
netbios_session_message:	dns_AAAA_reply:
dhcp_offer:	http_all_headers:
ssh_client_version:	ssl_client_hello:

Sampling of some native events

Intrusion Detection In-Depth

As you have learned, Bro comes with a full complement of scripts to perform analysis of traffic and process events. We mentioned previously that Bro is described as being "event-driven". The Bro developers defined event conditions and coded those events that they considered noteworthy and potential desired entry points for user-written custom scripts. This does not necessarily mean that an event is malicious; it is just an occurrence that the developers believe may be worth recording or assessing perhaps for further scrutiny or in conjunction with other activity or events.

Some of the many Bro events are listed above just to give you an idea of the volume and variety. The event names and definitions are found in the file "PREFIX/share/bro/base/bif/event.bif.bro" when not associated with a specific protocol. Protocol-specific event names and definitions are found in "PREFIX/share/bro/base/bif/plugins/Bro*bif.bro" directories where "*" is typically a protocol name or in a file "PREFIX/share/bro/base/protocols/protocolname/main.bro" where *protocolname* is one of the protocols like "ssl" or "http", etc. Looking at the slide list you can see some of the protocols Bro analyzes – IPv4, IPv6, TCP, UDP, SMTP, NETBIOS, DHCP, SSH, teredo, ICMP, FTP, DNS, HTTP, and SSL. As you can see by the names of the events, they represent the state of Bro, protocol states, as well as different aspects of the protocol.

Many of the events allow you to identify a given protocol, extract parts of that protocol, and perform some kind of processing on values in the protocol. For instance, "ssh_client_version" allows you to examine the SSH version number, perhaps compare the version number found, and say it is an obsolete version – perform some kind of activity that generates output. This can be e-mail that details the pertinent parts of the packet – perhaps the IP address of the client – to inform of the out-of-date version.

You must first find an event that triggers on an appropriate condition when you create a script and identify this event in your script. When the event conditions are met your script will be executed. Suppose you want to do some kind of processing if a particular IP address is seen. One option is to use the "new_connection" event and create a script that matches the IP field value from the connection, passed to it by the triggered event with your designated IP address, and then does some post-processing to inform you of the activity upon matching conditions.

Step 1: Need to Look for Event Name Associated with Bro Startup

more PREFIX/share/bro/base/bif/event.bif.bro

```
##! The events that the C/C++ core of Bro can generate. This is mostly
##! consisting of high-level network events that protocol analyzers detect,
##! but there are also several general-utility events generated by internal
##! Bro frameworks.
```

```
(snip)
```

```
## Generated at Bro initialization time. The event engine generates this
## event just before normal input processing begins. It can be used to execute
## one-time initialization code at startup. At the time a handler runs, Bro will
## have executed any global initializations and statements.
```

```
global bro_init: event();
```

Intrusion Detection In-Depth

The file that contains generic (not protocol-specific) Bro events is found in "PREFIX/share/bro/base/bif/event.bif.bro". You can peruse it from top to bottom to find an event that has a name that may be a good candidate for what you want to do. Each event is preceded by a description of what it does to assist you in your search.

If you want a list of all events available without going through the file line by line, execute the following:

```
grep "event" event.bif.bro | grep -v "##"
```

This searches the event file for the word "event", excluding any line with a comment in it. We don't want to see comments that talk about an event, just the event name itself. You can easily search through this list for an event that has a name similar to what you might want to do. If you find something that seems a likely match, you can read the associated description.

We've found an event called "bro_init" with a description of "one-time initialization code at startup". This seems to reflect exactly what we want to do – execute something after Bro begins.

Run the Script

```
event bro_init()
{
    print ("Started bro");
}
```

```
root@user:/tmp/broscripts# bro -r cmdexe.pcap broinit.bro
Started bro
```



Let's say we call our script "broinit.bro". Scripts supplied in Bro end with the extension of ".bro", however this isn't required. The script simply has the event name "bro_init" found in the file "event.bif.bro" that will cause the script to be invoked if the code path encounters it. As mentioned, scripts can have parameters passed to them, denoted between the parentheses following the event name. The "bro_init" event has no parameters. We place a Bro print statement that says "Started bro" as our single line of code.

Next we run Bro using with the name of the Bro script to run reading in cmdexe.pcap, although we don't need a pcap for this particular script, we include it to represent a more conventional situation. As you see, we get the output of "Started Bro".

Inform About a New Connection

```
user@user:/tmp/broscripsts# more broif.bro
```

```
event new_connection(c: connection)
{
  if (c$id$orig_h == 192.168.11.62 && c$id$resp_p == 80/tcp)
    print fmt("New Connection => orig: %s %s resp: %s %s",
              c$id$orig_h, c$id$orig_p, c$id$resp_h, c$id$resp_p);
}
```

```
user@user:/tmp/broscripsts# bro -r http.pcap broif.bro
```

```
New Connection => orig: 192.168.11.62 19086/tcp resp: 173.194.73.106 80/tcp
```



Now that you have a general idea how to run a script and understand the output that it generates, let's write a script that is more practical and has a bit of logic involved. According to the comments in "event.bif.bro", the "new_connection" event is described as follows "Generated for every new connection. This event is raised with the first packet of a previously unknown connection".

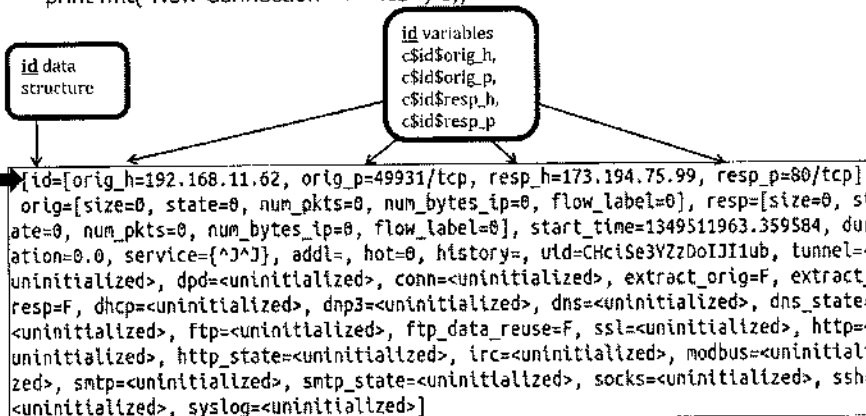
We will defer a discussion of the script code and the unique variable names for the next couple slides.

When we find a new connection with the source IP address of 192.168.11.62 and a destination port of TCP 80, we use a formatted print statement much like ones used in C and other programming languages where the first part describes the text output and the output format of the associated variables. The "%s" denotes string output for the source IP/port and destination IP/port of the variables that follow in the second part of the formatted print statement. When the script is run using "http.pcap" as input, we see the output of the script on the bottom of the slide.

It should be mentioned that Bro documentation is clear that it uses the terms originator and responder to reference the source and destination. They avoid the use of source and destination so that is why you see the variable names with "orig" and "resp" in them. We'll still use the terms of source and destination since we've become accustomed to doing so.

Connection Data Structure

```
print fmt("New Connection => %s ", c);
```



Intrusion Detection In-Depth

You will be exposed to the concept of a Bro connection when examining output and writing your own scripts so it is necessary to understand the connection data structure. The script on the previous slide generates a formatted print of a new connection known in Bro as the variable "c". Bro dumps the "c" data structure as seen in the slide.

A connection is a nested data structure much like we saw in the event signature `_match` script parameter "state". In order to identify a given variable, you must begin at the base structure "c" and identify any other nested structures. We see that the "id" data structure contains the variables to identify the source and destination IP addresses and ports. For instance "c\$id\$orig_h" identifies the source IP address. As you will recall, the "\$" serves to dereference or delimit each of the nested data structures as well as any variable in the final nested data structure.

Let's Examine the Script

```
event new_connection(c: connection)
{
    if (c$id$orig_h == 192.168.11.62 && c$id$resp_p == 80/tcp)
        print fmt("New Connection => orig: %s %s resp: %s %s",
            c$id$orig_h, c$id$orig_p, c$id$resp_h, c$id$resp_p);
}
```

The "new_connection" event passes connection data in an argument named "c" with a Bro type of connection. This is detailed where the event is defined in "event.bif.bro". The above script checks the connection to see if the source IP address "c\$id\$orig_h" is 192.168.11.62 and the destination port "c\$id\$resp_p" is TCP 80. First, how did we know the names of the variables that Bro associates with source IP address and destination port address? There is a file in "PREFIX/share/bro/base/init-bare.bro" that contains the names of all of the Bro's variables associated with any connection.

You'll notice the format of each variable used in this script begins with "c\$id\$". The "c" represents the connection data passed to the function; the "id" represents Bro's unique identification, UID, for this particular connection. The "c" connection is the base data structure, followed by an "id" data structure that is followed by all the relevant fields. The fields for identifying any new connection are the source IP is known as "orig_h", the source port as "orig_p", the destination IP address as "resp_h" and the destination port as "resp_p".

When we find a new connection with the source IP address of 192.168.11.62 and a destination port of TCP 80, we use a formatted print statement to display the output.

Put a New Script Into Production to Sniff Live Traffic

- Place your script file in PREFIX/share/bro/site directory or subdirectory you create
- Copy your script to this directory
- Edit local.bro to load your new script
 - Add a @load line with the name of the script to load
- Restart Bro loading new script

```
[BroControl] > install
[BroControl] > restart
```
- Go to directory PREFIX/logs/current
 - Examine file loaded_scripts.log to see if the script you loaded is there
- Generate some traffic to cause the script to fire
- Examine notice.log in PREFIX/logs/current -> PREFIX/spool/bro

Intrusion Detection In-Depth

Let's say we've got a proof of concept that our script works by invoking it from the command line in readback mode. What if you wanted to place this script into production to use as Bro is examining live network traffic? There are several tasks you must perform to let Bro know of the existence of a new script.

First, Bro expects any site-specific scripts to reside in "PREFIX/share/bro/site". Place your scripts there or create a subdirectory there to keep your personal scripts together.

As you know, the file "PREFIX/share/bro/site/local.bro" is the entry point into Bro's guide of what site-specific scripts exist. You must identify the file name of your script. You need to add a line with the format of "@load filename" of the script you want to load.

Next, you need to inform the running Bro process to install the new script and restart to include the new script. This is done in "broctl" using the "install" and "restart" commands.

There are two ways to discover whether or not your script was loaded. The first is a command in "broctl", "scripts" that lists all loaded scripts. This can produce a lot of output that scrolls by rapidly. The other way is to go to the current log directory "PREFIX/logs/current" and examine the "loaded_scripts.log".

Now test the script by generating some traffic to make it trigger. Finally, if the script raises a notice, examine the "notice.log" for the Notice generated by the new script. This resides in "PREFIX/logs/current" that has a symbolic link to "PREFIX/spool/bro".

Customization Option: Bro Notice Framework

- Permits your customized script to notify of activity and determine what to do with it
- Raises a notice informing user of activity along with description and traffic characteristics
- Notice policy describes actions that can be taken once notice raised
- Very sophisticated support, some beyond the scope of the class

Intrusion Detection In-Depth

Another feature of the policy neutral philosophy of Bro is the capability to customize notices for your site's needs. Bro deliberately does not use the word "alert" as found in Snort rules since the connotation is that something is amiss. The notice framework is what permits you to assign meaning and an assessment of importance to any activity. Additionally, it permits you to take one of several actions post detection.

"Raising a notice" is the term that Bro uses when you create a notice. There is extensive support for raising notices; however, we will be covering the more rudimentary aspects. We will learn to raise a notice and customize it by supplying it a name of our choosing and defining the output from the notice.

While it may seem like the coverage we provide of the notice framework is complete, we are actually doing a basic introduction. There is more documentation on the notice framework found at:

<http://www.bro.org/sphinx-git/frameworks/notice.html>

As with scripts and signatures, the best way to understand how they work is to experiment on your own. The information supplied in these sections is intended to give you the foundation to do that.

Example Notice

```
event bro_init()
{
  NOTICE({$note=Weird::Activity,
          $msg="My NOTICE message"});
}
```

An existing or created \$note value

Your informative \$msg message

```
user@host:/tmp/broscripsts: bro broinit-notice.bro
```

```
cat notice.log | bro-cut note msg actions
```

note	msg	actions
Weird::Activity	My NOTICE message	Notice::ACTION_LOG



Intrusion Detection In-Depth

broinit-notice.bro

Why do you even need to use something called a Notice Policy? Our other scripts created some informational output to the screen. While this is helpful in a debug or special run situation, it is not especially useful in production mode. Bro allows you to raise a notice that places all your output into "notice.log", making it more convenient to locate and further process the notices, if desired.

It is probably best to start with a simpler example since that will most effectively clarify what is known as the Notice Policy. As usual, we need to anchor off of a known Bro event since a notice is called from a Bro script. We are going to use "bro_init" again only because the event is always executed upon invocation of Bro regardless of the content of the traffic. The notice and message that will be generated from our script do not pertain to any traffic so no pcap is supplied. Again, this is used as an example just to start off simply, let you see how it works, and demonstrate the constructs involved.

Inside our script that gets executed when the "bro_init" event occurs, we define our Notice Policy using "NOTICE([variable=value, etc.])". It has many different parameters it can accept - see <http://www.bro.org/sphinx-git/frameworks/notice.html#raising-notices> for a complete discussion. Minimally, you are required to assign a value to the variable "note", and it is most informative to assign a value to "msg", much as you would do in a Snort rule, to create some output that reflects the nature of the activity. After all, that is pretty much the reason to use the Notice Policy in the first place. If this notice pertained to actual traffic, you could include traffic characteristics such as source and destination IP's and ports that will appear in an entry in "notice.log" where notices are placed by default.

We assign a value of "Weird::Activity" to the variable "note" and a value of "My NOTICE message" to the "msg" variable. The "msg" variable is straightforward; you assign it a value that you think best describes the activity that raises the notice.

The "note" variable is a required field in the Notice Policy definition to represent an existing or user-defined notice type. Suffice it to say that "Weird::Activity" is a default "note" value. The "note" variable value and concept are not so easy to describe. If you would like to delve deeper into this, reference the raising notices website link on the previous page. As well, there is a slide in the Bro Appendix with the title of "Bro-Defined Notice note Variable Value". We run Bro to trigger the event to raise the notice.

The "notice.log" has many different fields, although many are blank in this example, so we display the pertinent ones using bro-cut. We see that the note value in the log is "Weird::Activity" that represents a combination of the module and notice description that caused the notice to be raised. This will be shown in more detail on the next slide. As well, we see the message we created and the associated action "Notice::ACTION_LOG" with this notice, indicating that it should be recorded in "notice.log".

Notice Policy Action

- Apply an action to a notice
 - NOTICE::ACTION_LOG - default if unspecified, log the notice
 - NOTICE::ACTION_EMAIL - log and e-mail
 - NOTICE::ACTION_ALARM - log separately and e-mail contents on an hourly basis

Intrusion Detection In-Depth

A Notice Policy can have an action other than the default of logging to "notice.log". The above list represents the main Notice Policy actions available to perform when a notice is raised. There are others, but these are the most often used.

As you have seen, the default action is to place the notice in Bro's "notice.log" if no action is explicitly assigned. There is also an action of logging and e-mailing the notice. This assumes that you've configured e-mail recipients in "PREFIX/etc/broctl.cfg" and you need to have sendmail or some kind of mail package installed to support SMTP. A final option is to e-mail and log, but on an hourly basis as opposed to when the notice is raised.

We won't demonstrate how to assign a Notice Policy action since it is rather involved. Take a look at the Bro documentation <http://www.bro.org/sphinx-git/frameworks/notice.html#raising-notices> for the exact details.

Customization: Real-World Analysis Using Bro

Intrusion Detection In-Depth

Bro is very versatile and at times difficult to grasp when examined purely in theory. Therefore, this section is intended to assist you understand how Bro can be used to examine some real-world traffic.

Example 1: Basic Processing

```
jnovak@judy: /tmp/test-bro-basic$ bro -r sample1.pcap
conn.log dns.log http.log packet_filter.log weird.log

jnovak@judy: /tmp/test-bro-basic$ bro -r sample1.pcap -f 'udp port 53'
conn.log dns.log
```



Now that you have some background on the theory of how Bro works, let's begin with `sample1.pcap` to see how you might analyze the traffic using Bro. We begin simply by running Bro to read `sample1.pcap`. Note that we are running this in the directory `/tmp/test-bro-basic` that is our working directory. Log files are created in this directory so make sure that when you run Bro in readback mode like this to isolate it with its own directory. Otherwise your log files will get mixed in with whatever files are in your currently working directory.

Bro generates separate log output based on the nature and the protocols observed in the traffic. A connection log named `"conn.log"` is always created if connections are observed. A `packet_filter.log` is always generated; it contains the BPF statement used to filter traffic. By default this is, `'ip or not ip'`, but this can be altered as we will examine next. The `"dns.log"` contains records about DNS transactions, and as you would expect, the `"http.log"` contains records about HTTP traffic.

Most traffic logs have a field known as `"uid"` that represents a unique ID for every different connection observed by Bro. Bro may generate records in various logs for any given connection depending on the protocols found in that particular connection. Each record in every log associated with that connection maintains that same `"uid"`, therefore it is a good way to examine all the logs for records affiliated with that connection.

The `"weird.log"` has records of what Bro considers abnormal activity. For instance, this particular `"weird.log"` has records about a bad TCP checksum and also ones for `"data_before_established"`. The latter may not be accurate. When this pcap is examined in Wireshark for the first session assigned `"data_before_established"`, it is a properly established session with data sent and received.

You can run Bro with any BPF statement using the `"-f"` command line switch in this case we elect to view UDP port 53 traffic – typically DNS. The output logs now reflect that.

Thanks and attribution to Ismael Valenzuela for supplying `sample1.pcap` (along with several others) on his blog <http://blog.opensecurityresearch.com/2014/03/identifying-malware-traffic-with-bro.html>.

Examine conn.log

```
jnovak@judy:/tmp/test-bro-basic$ cat conn.log | bro-cut -d ts uid id.orig_h id.orig_p id.resp_h id.resp_p prot
| more
2014-03-07T07:35:57-0500 Cd5IYfNIXerRxLln5 172.16.88.10 49387 172.16.88.135 80 tcp
2014-03-07T07:35:59-0500 Cd6qZSFHqLOhKryah 172.16.88.10 49388 172.16.88.135 80 tcp
2014-03-07T07:35:59-0500 CA9Ev81TYbu2J47Cf1 172.16.88.10 49388 172.16.88.135 80 tcp
2014-03-07T07:36:01-0500 CY8VrnRuBAAFdlie 172.16.88.10 49389 172.16.88.135 80 tcp
2014-03-07T07:35:57-0500 CtUhtE3y7zx1YSpJ4 172.16.88.10 49387 172.16.88.135 80 tcp
2014-03-07T07:36:02-0500 CvGhjK107uZjXkeIB9 172.16.88.10 49389 172.16.88.135 80 tcp
2014-03-07T07:36:04-0500 C3X8fM1emqFnkZ4hVc 172.16.88.10 49391 172.16.88.135 80 tcp
2014-03-07T07:36:00-0500 CyfnXw4YbYQwvu4Lc4 172.16.88.10 49388 172.16.88.135 80 tcp
2014-03-07T07:36:04-0500 ClnGw22ScvqUMILoe8 172.16.88.10 49391 172.16.88.135 80 tcp
2014-03-07T07:36:06-0500 CKTp3u15ctDARxLaKa 172.16.88.10 49393 172.16.88.135 80 tcp
2014-03-07T07:36:02-0500 CDJcnK2FwkWV4tJg07 172.16.88.10 49389 172.16.88.135 80 tcp
2014-03-07T07:36:07-0500 C3FoJq2LDHSwL4RItf 172.16.88.10 49393 172.16.88.135 80 tcp
2014-03-07T07:35:59-0500 CtBGJy4eT2nYtJVLth 172.16.88.10 57268 172.16.88.135 53 udp
```

Intrusion Detection In-Depth

Each log has different field names and associated values. Some logs share many of the same values like source and destination IP's and ports. If you were to display a log file using a command like "cat" or "more" the contents would be displayed on the screen, however it would be difficult to read because there are many fields and values and most lines wrap because of the length of a given record.

The logs are TAB delimited, making them more readable when using the bro-cut command. Much like the Unix "cut" command, the output is displayed in columns. How do you know what the field names are to feed to the bro-cut command to display specific columns? They are listed on the top of each log – not a very user friendly approach. You have to use a display command like "cat" or "head" to expose the field names first and then run bro-cut to display the associated values using those field names.

The "-d" command line option displays the time in readable format. The time, uid, source and destination IP's and ports, and the protocol are displayed. The connection uid is a randomized string that always begins with "C".

Examine dns.log

```

jnovak@judy:/tmp/test-bro-basic$ cat dns.log | bro-cut id.orig_h id.orig_p id.resp_h id.resp_p query
172.16.88.10 57268 172.16.88.135 53 m69e31iwtscth14c49hwcyLxbyoti9gxoxlu.info
172.16.88.10 68736 172.16.88.135 53 kvm49myard66l48lynre21hqfun26a47hyn28kq.org
172.16.88.10 56844 172.16.88.135 53 htj56h34ewmzh44izn38nwcvg23bsb58irg63b18.net
172.16.88.10 52578 172.16.88.135 53 asi55f32nyernygxjsbqk27pyewcygzo21ps.com
172.16.88.10 53812 172.16.88.135 53 mydvxhdvh54135ayc69nroyh54drnqcvpoz.ru
172.16.88.10 64374 172.16.88.135 53 lubqlze11bvovgub68jrazhxaqmwhrkqj46.com
172.16.88.10 57825 172.16.88.135 53 gqe21muf32evntdvasd10j26k27pqlrptosgx.net
172.16.88.10 60943 172.16.88.135 53 kyoqpxg53nuf42g43oqo21l48a17d40c31k67j16h44.org
172.16.88.10 49742 172.16.88.135 53 teredo.ipv6.microsoft.com
172.16.88.10 53498 172.16.88.135 53 nxhyasg43a47exhum19g23f52fro21byayk57fs.info
172.16.88.10 62762 172.16.88.135 53 dsxgygrnud50pzj36hwpqazdrg43eyl38f12.blz
172.16.88.10 60577 172.16.88.135 53 axgql48nql28h34k67fvnylwo51csetj16qzcx.ru

```

Intrusion Detection In-Depth

Now, we'll examine the "dns.log" file – specifically the source and destination IP addresses and ports and the particular DNS query names. This can provide some insight into where the sender wishes to go. As you can see, these are some very strange DNS query names. Except for the teredo query, all appear to be long randomized strings followed by some Top Level Domain (TLD).

Strange DNS Name Queries

```
jnovak@judy:/tmp/test-bro-basic$ cat dns.log | bro-cut query | sort -u
a37fwf32k17gsgylqb58oylZgvlsi35b58m19bt.com
a47d20ayd10nvkshqn50Lrltgqcx68n20gup62.com
a47dxn60c59pziulsozaxm59dqj26dynvfnw.com
a67gwkaykulxczeueqf52mvcue61e11jrc59.com
ar1qo41oz128o51bsptk27atmzpzj66kzhvdq.com
asi55f32nyernygxjsbqk27pyewcygo21ps.com
atnzaxk27pxl28l28i15dvc39kzevbwdqe61fq.com
avh14cwg13b68jqlrf32bufwg33f42dzo51jqlx.net
awa57huhwj46ozhto11drbzo51c19m29btcqp42.org
axgql48mq128h34k67fvnylwo51csetj16gzcx.ru
ayp52n49msmwmthxoslwpxg43evg63esmreq.info
azg63j36dyhro61p32brgyo21k37fqh14d10k37fx.com
azn50i35btls148g33nre41g43ism39exc49lwn30.biz
```

Intrusion Detection In-Depth

Extracting only the name queries with bro-cut and sorting all the unique ones, we see that the queries are similar in that they are a randomized string with a length of 32-48 characters long.

The Gameover Zeus variant known as Murofet/Licat uses something known as a Domain Generation Algorithm (DGA) to make takedown attempts harder. This enables the command and control server to be decentralized. The domain names generated by Gameover Zeus produced a random string of letters and numbers of between 32-48 characters in length. Additionally, the TLD's of ru, com, biz, info, and net were used exclusively.

As you see, the discovery of the Gameover Zeus botnet was fairly easy with some rudimentary Bro processing.

Example 2: Find Heartbleed Attack Using Bro

```
judy@judy: /tmp/test-bro-heartbleed$ bro -r sslheartbleed.pcap

conn.log files.log notice.log packet_filter.log ssl.log x509.log

cat notice.log | bro-cut id.orig_h id.orig_p id.resp_h id.resp_p note msg
192.168.11.1 54848 192.168.11.128 443
Heartbleed::SSL_Heartbeat_Attack An TLS heartbleed attack
was detected! Record length 3, payload length 16384
192.168.11.1 54848 192.168.11.128 443
Heartbleed::SSL_Heartbeat_Attack_Success An TLS heartbleed
attack detected before was probably exploited. Transmitted
payload length in first packet: 16384
```



Intrusion Detection In-Depth

sslheartbleed.pcap

Let's explore how Bro can be customized via its scripting capabilities to detect the heartbleed TLS/SSL attack. If you recall from Day 3, the heartbleed attack sends a malformed heartbeat message in the client "hello" causing the server to return a heartbeat reply with data from a memory leak that can expose the server's private key, session tokens, as well as other data such as usernames and passwords.

A Bro developer named Bernhard Amann created a script to detect the heartbleed attack. This runs on versions of Bro 2.2 and later. Once run, it generates several log files, the one of interest to us is the "notice log". This file stores raised notices generated by a Bro script that uses the "NOTICE" facility to generate a message. If you recall, these can be informational messages that the script developer creates and stores, but more likely, they are indications of activity that is noteworthy in some way.

We use the bro-cut command to list fields and associated values of interest. Remember that the unique field names referenced in the bro-cut command are found at the top of every log. The fields "note" and "msg" found specifically in the notice log show that a heartbleed attack was detected and that was probably successful.

We'll pursue what the script was looking for, what it found, and how it generated a message for "notice.log".

The pcap used for this example was downloaded from a blog maintained by Didier Stevens, a well-known and very well respected researcher in the field:

<http://blog.didierstevens.com/2014/04/09/heartbleed-packet-capture/>

Thanks and attribution to Didier Stevens for supplying this pcap.

Under the Hood of a Script

Excerpts from script heartbeat.bro

```
event ssl_heartbeat(c: connection, is_orig: bool, length: count, heartbeat_type:
    count, payload_length: count, payload: string)
```

checks to see if there is a heartbeat request type 1, examines length for threshold value

checks to see if there is a heartbeat reply message type after determination of non-standard heartbeat request

```
NOTICE([$note=SSL_Heartbeat_Attack,
    $msg=fmt("An TLS heartbleed attack was detected!
Record length %d, payload length %d", length, payload_length),
```

```
NOTICE([$note=SSL_Heartbeat_Attack_Success,
    $msg=fmt("An TLS heartbleed attack detected before
was probably exploited. Transmitted payload length in first packet:
    %d", payload_length)
```

Intrusion Detection In-Depth

We do not intend to make you an expert Bro scripter from this course, however, knowing the underlying structure of the heartbleed scripts will assist you in understanding them as well as helping you write your own should you choose.

How does Bro processing get to that event in the first place? The heartbleed packet requires Bro SSL processing once Bro determines that TCP port 443 is used. Then there are different scripts associated with SSL processing, in this case one called "heartbleed.bro", that may get executed. If an SSL heartbeat message is present in the SSL exchange, the event named "ssl_heartbeat" will examine the heartbeat to first determine if a heartbeat message of type 1 representing a heartbeat request is present. It continues heartbeat heartbleed processing to examine the heartbeat request payload length for greater than some threshold value and raises the first notice if found. Next, it examines the traffic for a heartbeat reply. This pcap has a heartbeat reply causing the second notice to be raised.

Example 3: Using Bro to Extract Files from pcap Payload

```
bro -r php-malware.pcap extract.bro
conn.log dhcp.log dns.log extract_files files.log http.log packet_filter.log
reporter.log
```

```
more extract.bro
event file_new(f: fa_file)
{
    Files::add_analyzer(f, Files::ANALYZER_EXTRACT);
}
```

```
cat files.log | bro-cut -d fuid tx_hosts rx_hosts conn_uids source mime_type
extracted | head -1
FXx5sS2t8h0zFGNv3a 69.147.83.199 192.168.40.10
CGo2fX3tTEL5I8Aue7 HTTP text/html extract-HTTP-
FXx5sS2t8h0zFGNv3a
```



Suppose you have a pcap where you know, suspect, or would like to know if there were any files transferred and be able to examine those files. It is possible to extract files in Bro versions 2.2 and later.

By default, Bro will not do this unless you supply it a script to do so. We will supply the script name, "extract.bro" when we use Bro in readback mode, however, if you wanted to do the same for Bro in production mode sniffing off the interface, this script or something like it could be loaded into the list of bro start-up scripts. The script above is named "extract.bro" that triggers off an event known as "file_new" to extract the file to a subdirectory called "extract_files" where each filename in the directory contains a uniquely generated file id (fuid) beginning with the letter "F". The file name begins with the word "extract", followed by the protocol in which it was found (HTTP in this case), followed by the fuid.

A log named "files.log" is generated that contains data associated with the file such as the fuid, the sender and receiver, the connection id (uid) associated with the session in which the activity occurred, the MIME type that indicates the type of data found in the file, and the file name associated with the file in the "extract_files" subdirectory.

Looking at the output of the first line of "files.log", you see the fuid "FXx5sS2t8h0zFGNv3a", the sending IP of 69.147.83.199, the receiving host of 192.168.40.10, the connection id (uid) of "CGo2fX3tTEL5I8Aue7", the source protocol of HTTP, a MIME type of "text/html" and the associated file name in subdirectory "extract_files".

The pcap used for this example was downloaded from a link from Barracuda Labs:

<https://barracudalabs.com/2013/10/php-net-compromise>

Barracuda Labs performs research and some security tools. Gratitude and attribution to them for supplying the pcap.

Examine HTTP Traffic Log

```

cat http.log | bro-cut id.orig_h id.orig_p id.resp_h id.resp_p host uri resp_mime types
192.168.40.10 1043 144.76.192.102 80 zivvgmyrwy.3razbave.info /
b0047396f70a98831ac1e3b25c324328/8fdc5f9653bb42a310b96f5fb203815b.swf text/html
192.168.40.10 1044 144.76.192.102 80 zivvgmyrwy.3razbave.info /
b0047396f70a98831ac1e3b25c324328/b7fc797c851c250e92de05cbafe98609 text/html
192.168.40.10 1042 144.76.192.102 80 zivvgmyrwy.3razbave.info /?
695e6cca27beb62ddb0a8ea707e4ffb8=43 -
192.168.40.10 1048 144.76.192.102 80 144.76.192.102 /?9de26ff3b66ba82b35e31bf4ea975dfe
application/x-dosexec
192.168.40.10 1049 144.76.192.102 80 144.76.192.102 /?90f5b9a1fbc2e4a879001a28d7940b4
application/x-dosexec
192.168.40.10 1050 144.76.192.102 80 144.76.192.102 /?8eec6c596bb3e684092b9ea8970d7eae
192.168.40.10 1051 144.76.192.102 80 144.76.192.102 /?35523bb81eca604f9ebd1748879f3fc1
application/x-dosexec
192.168.40.10 1052 144.76.192.102 80 144.76.192.102 /?b28b06f01e219d58efba9fe0d1fe1bb3
application/x-dosexec
192.168.40.10 1069 144.76.192.102 80 144.76.192.102 /?52d4e644e9cda518824293e7a4cdb7a1
application/x-dosexec

```

Intrusion Detection In-Depth

Now we look at "http.log" since the files generated were associated with HTTP. Specifically, we'll examine the source and destination IP's and ports, the hostname found in the HTTP request header, the requested URL, and the MIME type of the returned file. The records displayed above are ones of most interest to us.

Of note are the hostname in the first three records – something that looks like some randomly generated name "zivvgmyrwy.3razbave.info". Then take a look at some of the URL names that begin with a "?" followed by a series of random numbers and characters, all of the same length. And, most distressing is that the server returned a file that was determined to be a DOS Windows executable. Obviously, this needs to be investigated.

Which HTTP Connections Returned Executable Files?

```

cat http.log | bro-cut id.orig_h id.orig_p id.resp_h id.resp_p host url resp_mime_types uid resp_fuids |
grep dosexec
192.168.48.18 1848 144.76.192.102 80 144.76.192.102 /?9de26ff3b66ba82b35e31bf4ea975dfe
application/x-dosexec CQP4gv3w7pe4c4d13j FXJNH216c9M13Ubqkx
192.168.48.18 1839 144.76.192.102 80 144.76.192.102 /?99f5b9a1fbc2e4a879001a28d7940b4
application/x-dosexec Cch1ue3V7jkF04e6h1 FOkjyjicZw3URAkp9e
192.168.48.18 1851 144.76.192.102 80 144.76.192.102 /?35523bb81eca604f9ebd1748879f3fc1
application/x-dosexec CaL0YA3bDKTmcPrmua Fl2eCy4tt0Bq8Fb589
192.168.48.18 1853 144.76.192.102 80 144.76.192.102 /?b28b86f81e219d58efba9fe0d1fe1bb3
application/x-dosexec CNUJtqw1iUvg2cz7a Flfwkc15otYvmG1hf
192.168.48.18 1869 144.76.192.102 80 144.76.192.102 /?152d4e644e9cda518824293e7a4cdb7a1
application/x-dosexec Ch44tn1LAjJr3ZJKEh Fd7scMcGCaVzcyuac

```

Intrusion Detection In-Depth

Continuing with our search, let's extract all the HTTP sessions where an executable files was returned. We extract many of the same fields using bro-cut as we did in the previous slide, but now our interest is with the associated connection uid, and the file identification fuid. We want to see sessions only with a "dosexec" as part of the MIME type and use the grep command to filter those out for us.

What Activity Was Associated with a Given Connection uid?

```

grep CQP4gv3w7pe4c *
conn.log:1382470082.246961 CQP4gv3w7pe4c4d13j 192.168.40.10 1048 144.76.192.102 80
tcp http 1.011161 219 89367 SF - 73986 ShADadcFF 43 1947
14 13149 (empty)
files.log:1382470082.605224 FXJNn216c9Ml3Ubqkx 144.76.192.102 192.168.40.10
CQP4gv3w7pe4c4d13j HTTP 0 EXTRACT application/x-dosexec - 0.000000 -
F 1268 - 6926 0 F - - - HTTP-
FXJNn216c9Ml3Ubqkx.exe
files.log:1382470083.257683 FXJNn216c9Ml3Ubqkx 144.76.192.102 192.168.40.10
CQP4gv3w7pe4c4d13j HTTP 0 (empty) - - 0.000000 - F
0 69856 0 T - - - -
http.log:1382470082.418258 CQP4gv3w7pe4c4d13j 192.168.40.10 1048 144.76.192.102 80
1 GET 144.76.192.102 /?9de26ff3b66ba82b35e31bf4ea975dfe - Mozilla/4.0 (compatible;
MSIE 8.0; Windows NT 5.1; Trident/4.0) 0 7108 200 OK - -
(empty) - - - - FXJNn216c9Ml3Ubqkx application/x-dosexec

```

Intrusion Detection In-Depth

Let's examine the first session from the previous slide where an executable was downloaded. We again use the grep command to find the uid associated with that session "CQP4gv3w7pe4c" and see the "conn.log", "files.log" and "http.log" entries. We already knew about these; we didn't find any other activity associated with other protocols. Now, we can examine the nature of the file downloaded.

Examine the Extracted File Using xxd

```
xxd extract-HTTP-Fd7scMcGCaVzcyuac
```

```
0001b00: 0000 0000 0000 0000 0000 0000 0000 4d5a .....MZ
0001b10: 9000 0300 0000 0400 0000 ffff 0000 b800 .....
0001b20: 0000 0000 0000 4000 0000 0000 0000 0000 .....@.....
0001b30: 0000 0000 0000 0000 0000 0000 0000 0000 .....
0001b40: 0000 0000 0000 0000 0000 0000 e800 0000 0e1f .....
0001b50: ba0e 00b4 09cd 21b8 014c cd21 5468 6973 .....!.L!This
0001b60: 2070 726f 6772 616d 2063 616e 6e6f 7420 program cannot
0001b70: 6265 2072 756e 2069 6e20 444f 5320 6d6f be run in DOS mo
0001b80: 6465 2e0d 0d0a 2400 0000 0000 0000 3d4b de...$....=K
0001b90: d031 792a be62 792a be62 792a be62 62b7 .1y*.by*.by*.bb.
0001ba0: 2062 682a be62 62b7 1462 332a be62 62b7 bh*.bb..b3*.bb.
0001bb0: 1562 592a be62 7052 2d62 682a be62 792a .bY*.bpR-bh*.by*
0001bc0: bf62 202a be62 62b7 1162 782a be62 62b7 .b *.bb..bx*.bb.
0001bd0: 2562 782a be62 62b7 2462 782a be62 62b7 %bx*.bb.$bx*.bb.
0001be0: 2362 782a be62 5269 6368 792a be62 0000 #bx*.bRichy*.b..
0001bf0: 0000 0000 0000 5045 0000 4c81 0400 d5b4 .....PE.L.....
0001c00: 6552 0000 0000 0000 0000 e000 0301 0b01 eR.....
0001c10: 0a00 00d6 0000 0082 0000 0000 0000 581b .....X.
```

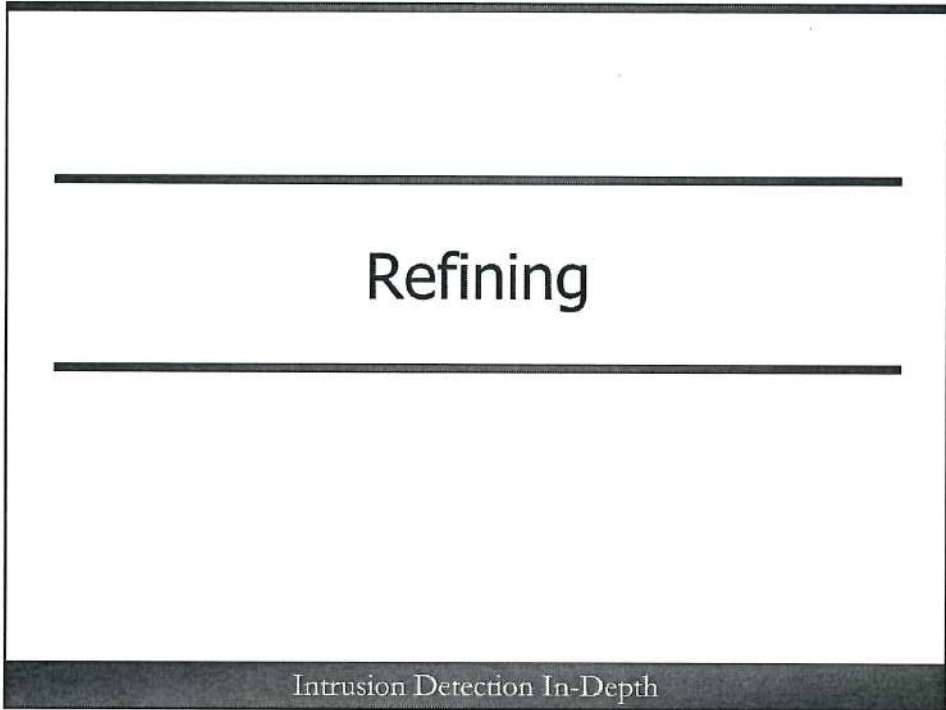
Intrusion Detection In-Depth

The linux xxd command can be used to display the hex content of a file and interpret any ASCII characters, much like tcpdump -X mode can do. What we see from the xxd output of one of the extracted executable files are indeed indications that an executable was downloaded.

The "MZ" is a magic number designation at the beginning of an executable that indicates that the file is an MS-DOS executable file format along with the "PE" that indicates it is a Windows Portable Executable. This is a standard MS-DOS header of a Win32 Portable Executable file. A 32-bit Windows PE file that is run in a 16-bit DOS environment generates an error message "This program cannot be run in DOS mode".

We'll stop our discussion here, but the extracted file could be scrutinized more carefully and possibly reverse engineered by someone familiar with the process. Bro was able to quickly expose this valuable insight using some rudimentary commands.

This traffic in the pcap is associated with an incident when php.net was hacked and downloading malware to unsuspecting visitors. Php.net is a site that distributes downloads for PHP.



This page intentionally left blank.

Bro-Specific Performance Factors

- Use of clusters for distributed packet capture and processing
- Number and types of scripts loaded
- Number and types of signatures loaded
- Value of `dpd_buffer_size` for signature inspection depth

Intrusion Detection In-Depth

As we discussed at length, Bro clusters offer the capability to do parallel processing by distributing the load among Bro workers to collect and process traffic, using the proxy to provide communications among the other Bro components, and the manager to orchestrate the whole operation. The cluster configuration is flexible, permitting the easy addition of new workers for performance improvements.

Bro comes with many default scripts that are loaded upon start of Bro. Bro optionally allows you to run site-specific as well as policy scripts. Policy scripts such as profiling can consume resources and should be used for short durations for the purpose of debugging. Obviously, the more optional scripts loaded, the more processing Bro must perform.

As well, the number and types of user-written signatures may have an effect on performance. As we learned, Bro uses regular expressions to find content for signatures. Bro can perform a generic search of all or part of the payload, depending on the `dpd_buffer_size` that designates how deep into the payload to search. Or Bro can perform content searches that are more constrained such as in HTTP headers where content may be found more rapidly. We discussed that `dpd_buffer_size` has a default value of 1024 bytes for more efficient processing. However, that may not be suitable if you believe that the content sought is deeper into the payload. Naturally, increasing this size requires Bro to do more inspection.

Examine Bro's Resource Consumption

- Load PREFIX/policy/misc/stats.bro – lightweight
- Load PREFIX/policy/misc/profiling.bro – debugging only
- Load PREFIX/policy/misc/capture-loss.bro – reports in notice.log
- Execute "capstats" in broctl

Intrusion Detection In-Depth

There are several methods to inspect Bro's resource consumption. The first three methods require you to load a Bro script. The first method is to use the script "PREFIX/share/bro/policy/misc/stats.bro". The script has the following description of its function:

"Log memory/packet/lag statistics. Differs from profiling.bro in that this is lighter-weight (much less information, and less load to generate). "

The /usr/local "PREFIX/policy/misc/profiling.bro" script supplies very detailed feedback, including memory allocated, number of connections, TCP states, number and activity of current threads, to mention a few. This script should be loaded for debugging purposes only because of processing and storage overhead.

Another optional script "PREFIX/share/bro/policy/misc/capture-loss.bro" can be used to capture packet loss. Logged records appear in the file "notice.log".

Finally, there is a "capstats" command available in the "broctl" interface that can monitor a network interface for a given interval of time and report statistics of traffic and dropped packets.

There are parallels loading policy scripts and enabling Snort's preprocessors or special configuration directives to do performance monitoring. Both have most performance monitoring capabilities turned off since they perform computations that consume resources.

More details on these processes can be found in the Appendix slides associated with Bro "Refining".

Updating

See Appendix of Bro Material for Updating material

Intrusion Detection In-Depth

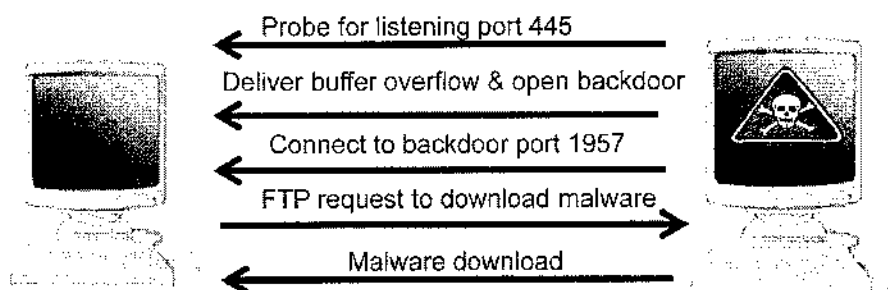
This page intentionally left blank.

Postscript: Snort and Bro Results from Analyzing the Same Traffic

Intrusion Detection In-Depth

This page intentionally left blank.

Analyze Attack Trace Activity Using Snort and Bro



Intrusion Detection In-Depth

We examined a pcap with some malicious activity by using Wireshark on Day 3. Let's see how you might examine this same activity using Bro.

Let's summarize the activity that occurred to remind you what transpired. First, the automated attack software probed the victim host on port 445. The software then attempted and successfully exploited a vulnerability in certain Active Directory service functions of the Local Security Authority Subsystem Service (LSASS) in Microsoft Windows.

A backdoor was opened on port 1957 of the victim. The victim connected to the attacker's FTP server to download some malware. Finally, the malware was downloaded to permanently infect the host and instruct it to perform malicious activity.

First, we'll show the Snort alerts from the traffic. Then we'll see how Bro might analyze the same traffic.

Snort Alerts from Same Traffic

Snort alerts:

```
snort -r attack-trace.pcap -q -A console -K none -c
/etc/snort/snort.conf
```

```
04/19-23:28:29.447746  [**] [1:2466:7] NETBIOS SMB-DS IPC$ unicode
share access [**] [Classification: Generic Protocol Command
Decode] [Priority: 3] (TCP) 98.114.205.102:1828 ->
192.150.11.111:445
04/19-23:28:30.172468  [**] [1:2514:7] NETBIOS SMB-DS DCERPC LSASS
DsRolerUpgradeDownlevelServer exploit attempt [**]
[Classification: Attempted Administrator Privilege Gain]
[Priority: 1] (TCP) 98.114.205.102:1828 -> 192.150.11.111:445
04/19-23:28:30.178588  [**] [1:648:7] SHELLCODE x86 NOOP [**]
[Classification: Executable Code was Detected] [Priority: 1] (TCP)
98.114.205.102:1828 -> 192.150.11.111:445
```

Intrusion Detection In-Depth

We saw this same slide in Day 3 with the alerts that Snort generated when fed that same traffic. It finds shell code NOP characters (0x90) as well as finding the signs of the LSASS vulnerability. We can conclude that Snort did a good job of finding the malicious traffic.

Now, let's see how Bro handles this same traffic.

A Couple of Things to Keep in Mind When Performing Bro Log Correlation

- A connection log entry should appear for all initial TCP/UDP connection activity
- Many logs share common fields that can be used in correlation
 - Source IP/port
 - Destination IP/port
 - Timestamp
- A shared UID record among the logs means that the reported activity occurred in a single session

Intrusion Detection In-Depth

Before exploring how Bro analyzes this same traffic, it is helpful to know what is available in the generated logs that can be used for correlation. The "conn.log" contains a record for each TCP or UDP connection. The TCP connection is recorded even if the three-way handshake is not completed. This log also contains the number of packets and bytes per conversation side. You will see when we cover SiLK, an open source network flow program, on Day 5 it too records similar data.

Bro's capability to correlate among captured logs can be assisted by using the command `bro-cut` to cut the tab delimited fields by field name. Knowing other Unix commands such as "sort", and "uniq" (discussed on Day 6) can greatly enhance your ability to correlate the output. Any log that contains traffic records has a timestamp and data about source and destination IPs and ports. There are informational logs such as the "notice_policy.log" and "packet_filter.log" that reflect the configuration of Bro at the time of run that do not contain traffic entries.

The UID is a shared generated value reflected in log entries for a particular session. You may see records with the same UID in the "conn.log", a protocol log, and perhaps "notice.log" or any other log for that stream. Keep in mind that traffic of interest may have several different associated sessions so it may not be possible to discover all the activity from a given attacker by using a single UID.

FTP File Retrieval Notice

more mynotices

```
#redef dpd_buffer_size = 10000 &redef;  
#uncomment above statement for Bro to search 10000 bytes into packet/stream in  
signature
```

```
module FTP;
```

```
event ftp_request(c: connection, command: string, arg: string)
```

```
{  
  if (command == "RETR")  
    NOTICE([$note=Weird::Activity, $msg="Found FTP retrieve file command",  
            $conn=c]);  
}
```



Intrusion Detection In-Depth

mynotices

First, let's prepare a script and signature. The script is not necessarily for the LSASS exploit, however the signature is.

Let's say that we have an existing script that raises a notice any time that FTP file retrieval occurs. The file named "mynotices" contains the code to raise a notice. It sets up a module that we call "FTP". We'll use a default note type of "Weird::Activity" and write out the message of "Found FTP retrieve command" and the connection values to the "notice.log".

There is a Bro event named "ftp_request" that we use to invoke that code. Conveniently enough, it has a field known as "command" that examines the FTP command. This is exactly what we want. FTP translates a file "get" request into the "RETR" command. When this is detected we write the "notice.log" output.

We'll revisit the commented lines on the top of the script in a couple of slides.

Isass Signature

```
more attack.sig

signature Isass-from-445 {
  ip-proto == tcp
  src-port == 445
  payload /*Isass*/
  event "Isass activity"
}

signature Isass-to-445 {
  ip-proto == tcp
  dst-port == 445
  payload /*Isass*/
  event "Isass activity"
}
```



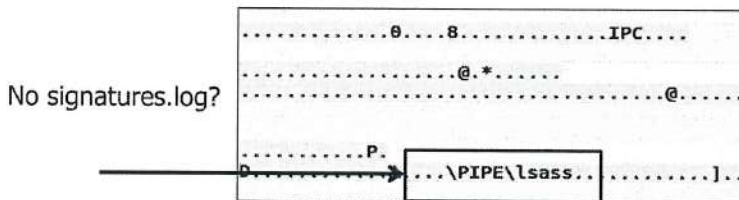
Further suppose we have been concerned that there are published exploits for the latest LSASS vulnerability. We know that LSASS is associated with TCP port 445 known as microsoft-ds. However, we don't know if the attack will be from or to port 445 and there is no way to designate both in a single signature so we write two signatures to cover both server and client traffic.

The signatures are found in a file named "attack.sig" that generates an event message in the "signatures.log" file whenever the string "Isass" is found anywhere in the payload. A message of "Isass activity" describes the activity.

This is not a good signature since the string "Isass" is commonly found in port 445 traffic; meaning there would be a lot of false positives. This signature was created merely for demonstration purposes.

Run Bro Using attack-trace.pcap

```
user@host:/tmp/broscripts: bro -s attack.sig -r attack-trace.pcap  
mynotices  
ls *log  
conn.log ftp.log notice.log notice_policy.log packet_filter.log weird.log
```



"By default Bro stops matching on a connection after seeing 1K of payload"
redef dpd_buffer_size = 10000;



Intrusion Detection In-Depth

attack.sig
attack-trace.pcap
mynotices

Now, let's fire up Bro for a test run before we load our script and signature. We run Bro on the command line using the `attack-trace.pcap` specifying the location of the signature and script file. We take a quick look at the logs generated from the run and find that there is a "notice.log", but there is no "signatures.log". Although not shown, the "notice.log" contains our notice with a message of "Found FTP file retrieve command". Yet, if we look at Wireshark's reassembled output from one of the sessions, we do indeed see the string "lsass".

Further scrutiny of Bro's behavior default reveals that Bro examines the first 1024 bytes of the reassembled stream only as indicated at <https://www.bro.org/sphinx-gjt/frameworks/signatures.html>. This is apparently for performance efficiency, however it is important to be aware of this when creating signatures. In order to change this default behavior, we need to redefine a value in the variable named "dpd_buffer_size" in the script. We select a value of 10,000 just to make sure that the signature fires this time. This redefinition was found as the first commented statement in the script file "mynotices" file. Although not shown above, when Bro is rerun a second time the statement is uncommented and a "signatures.log" is created with our message of "lsass activity".



Logs that are examined in the next set of slides are generated from running the Bro command on the slide. These logs are not supplied in your demo files.

Let's Examine Our Event Output

```
cat notice.log | bro-cut -d ts uid id.orig_h id.orig_p id.resp_h id.resp_p proto note  
msg
```

```
2009-04-19T23:28:30-0400      hD4RiDVCIRf  98.114.205.102 1828  
192.150.11.111 445      tcp      Signatures::Sensitive_Signature  
192.150.11.111: lsass activity
```

```
2009-04-19T23:28:34-0400      ZfUIGvkzs6e  192.150.11.111 36296  
98.114.205.102 8884      tcp      Weird::Activity Found FTP  
retrieve      file command
```

Intrusion Detection In-Depth

We use bro-cut to look at some selected "notice.log" data. We see that both the signature to find LSASS activity and the script to raise a notice upon seeing FTP file retrieval worked. These entries serve to inform us of noteworthy activity much like a Snort alert might do.

If you run Bro in production sniffing mode, this is one of the first log files to examine since it contains notices that have been raised that reflect notable activity. Entries in this log may have associated activity in other logs that may help assess what has transpired.

conn.log and ftp.log

```
.cat conn.log | bro-cut -d ts uid id.orig_h id.orig_p id.resp_h id.resp_p
-----
2009-04-19T23:28:28-0400 P6CAK1Xrzej 98.114.205.102 1821
192.150.11.111 445
2009-04-19T23:28:30-0400 2MmnxQ9dP2 98.114.205.102 1924
192.150.11.111 1957
2009-04-19T23:28:28-0400 hD4RiDVCIRI 98.114.205.102 1828
192.150.11.111 445
2009-04-19T23:28:33-0400 ZfUIGvkzs6e 192.150.11.111 36296
98.114.205.102 8884
2009-04-19T23:28:34-0400 y05ArTaWpL7 98.114.205.102 2152
192.150.11.111 1080

.cat ftp.log | bro-cut -d ts uid id.orig_h id.orig_p id.resp_h id.resp_p command arg
-----
reply_msg
2009-04-19T23:28:34-0400 ZfUIGvkzs6e 192.150.11.111 36296
98.114.205.102 8884 RETR ftp://98.114.205.102/./ssms.exe
Opening BINARY mode data connection
```

Intrusion Detection In-Depth

The "conn.log" and "ftp.log" were also generated. The "conn.log" is generated by default when any new connection is observed. The "ftp.log" was created because FTP is a protocol that Bro understands and tracks and was found in the traffic. We discover that we didn't even have to write a notice for the RETR command since the FTP decoder logs it by default. However, our script raised a notice that placed an entry in the "notice.log" (not shown) that is more likely to garner attention than the "ftp.log" output shown at the bottom. We see a download of the file "ssms.exe" in the "ftp.log".

The "conn.log" is comparable to Wireshark's Statistics for TCP Conversations. This is often where we start when examining some kind of activity in Wireshark. The "conn.log" provides a good overview of connections/conversations.

We see the entries created in "conn.log" from the traffic. Look at the next slide for more specific information on the connections.

Explanation of Some conn.log Fields

cat	conn.log	bro-cut	id.orig_h	id.orig_p	id.resp_h	id.resp_p	orig_ip_bytes	orig_pkts
	resp_ip_bytes	resp_pkts	conn_state	history				
98.114.205.102	1821	192.150.11.111	445	168	4	128		
3	SF	ShAFaf						
98.114.205.102	1924	192.150.11.111	1957	381	6	250		
6	SF	ShAdDaFf						
98.114.205.102	1828	192.150.11.111	445	4777	14	1590		
17	RSTO	ShADadR						
192.150.11.111	36296	98.114.205.102	8884	763	14	850		
12	SF	ShAdDCFaRf						
98.114.205.102	2152	192.150.11.111	1080	165088	159	4488		
112	SF	ShADaFf						

SF - Normal establishment and termination. Note that this is the same symbol as for state S1. You can tell the two apart because for S1 there will not be any byte counts in the summary, while for SF there will be.

RSTO - Connection established, originator aborted (sent a RST).

Intrusion Detection In-Depth

Continuing with the discussion of the fields displayed from the "conn.log", let's evaluate a few more that give some context about the connection, much like Wireshark does. You also can display data on source and destination bytes and packets to get a general idea of the size of the activity in the connection. Also, there are two fields `conn_state` and `history` that can assist you in understanding the TCP session state and session history.

Unfortunately these are not documented so the best place to look for the explanation of the fields is in the script source code:

```
"PREFIX/share/bro/base/protocols/conn/main.bro"
```

The `conn_state` column has two particular values for this set of connections - "SF" and "RSTO". These are only two of the many states possible. The "SF" reflects normal session establishment and termination. The "RSTO" indicates that the originating host sent a RESET to abort the connection.

Like the Wireshark conversation summation, we see an anomaly in the final entry in "conn.log". There were 165088 bytes sent by 98.114.205.102. That was what stood out in Wireshark that caused further investigation when we examined this same traffic on Day 3.

The history values are explained on the next slide.

Explanation of conn.log history Column Value

<u>ShAdDaFf</u>	<u>history</u>
98.114.205.102 .1924 > 192.150.11.111.1957: Flags [S], length 0	S
<u>192.150.11.111</u> .1957 > 98.114.205.102.1924: Flags [<u>S</u>],ack, length 0	h
98.114.205.102 .1924 > 192.150.11.111.1957: Flags [., ack , length 0	A
<u>192.150.11.111</u> .1957 > 98.114.205.102.1924: Flags [<u>P</u>],ack, length 1	d
98.114.205.102 .1924 > 192.150.11.111.1957: Flags [<u>P</u>],ack, length 123	D
<u>192.150.11.111</u> .1957 > 98.114.205.102.1924: Flags [., ack, length 0	a
98.114.205.102.1924 > 192.150.11.111.1957: Flags [<u>P</u>],ack, length 10	
192.150.11.111.1957 > 98.114.205.102.1924: Flags [.,ack, length 0	
192.150.11.111.1957 > 98.114.205.102.1924: Flags [<u>P</u>],ack, length 1	
98.114.205.102 .1924 > 192.150.11.111.1957: Flags [F],ack, length 0	F
<u>192.150.11.111</u> .1957 > 98.114.205.102.1924: Flags [<u>E</u>],ack, length 0	f
98.114.205.102.1924 > 192.150.11.111.1957: Flags [.,ack, length 0	

Intrusion Detection In-Depth

Probably the best way to explain the history values seen in this "conn.log" for a selected session is to view the tcpdump output for that same session. Output field values pertinent for this discussion are displayed from tcpdump. Look at the second connection found in "conn.log" with a history value of "ShAdDaFf". Any letter that is upper case represents the originator of the session and any letter that is lower case represents the recipient. For this discussion, 98.114.205.102 (bolded) is the originator and 192.150.11.111 (underlined) is the recipient for packets that have a history value. You see that any packet from 98.114.205.102 has a history value in upper case while traffic from 192.150.11.111 is represented in lower case.

Consult "PREFIX/share/bro/base/protocols/conn/main.bro" for full explanations of all possible values. The "S" indicates the originator sent a SYN; the "h" indicates that the recipient responded to establish a session. The "A" represents an acknowledgement by the originator, the "d" data sent by the recipient, "D" data sent by the originator, "a" an acknowledgement by the recipient, and the "F" and "f" - a FIN by the originator and recipient respectively. It appears that there is a single entry for each history state though there may be multiple instances of each in a session. There is a single entry for data sent and acknowledged although there are multiple packets representing that state.

This is a little esoteric at first, but it does convey the history of the connection very concisely. Using this in conjunction with the conn_state value for the connection, you get an abbreviated representation of the progress of the state throughout the session.

Search for Large Byte Transfer Using UID

```

grep ZfUIGvkzs6e *.log
conn.log:1240198113.457215      ZfUIGvkzs6e      192.150.11.111  36296
98.114.205.102  8884      tcp      ftp      11.13659177    214      SF
-              26      ShAdDCFaRf    14      763      12      850
(empty)
ftp.log:1240198114.384010      ZfUIGvkzs6e      192.150.11.111  36296
98.114.205.102  8884      1      <hidden>      RETR
ftp://98.114.205.102/./ssms.exe -      -      0      150
Opening BINARY mode data connection -      -
notice.log:1240198114.384010      ZfUIGvkzs6e      192.150.11.111  36296
98.114.205.102  8884      tcp      FTP::FTP_RETR Found FTP
retrieve file command -      192.150.11.111  98.114.205.102
8884      -      bro      Notice::ACTION_LOG 63600.000000      F

```

Intrusion Detection In-Depth

Let's examine the session with the large data transfer by its UID "ZfUIGvkzs6e". We use the Unix grep command to find all occurrences in any file that ends with ".log". We see activity in the "conn.log" and "ftp.log" as previously shown, along with the "notice.log".

Revisit conn.log

2009-04-19T23:28:28-0400	P6CAK1Xrzei	98.114.205.102	1821	①
192.150.11.111	ShAFaf	168 4	128 3	
2009-04-19T23:28:30-0400	2MmnxQ9dP2	98.114.205.102	1924	②
192.150.11.111	ShAdDaFf	381 6	250 6	
2009-04-19T23:28:28-0400	hd4RiDVCIRI	98.114.205.102	1828	③
192.150.11.111	ShADadR	4777 24	1590 17	
2009-04-19T23:28:33-0400	<u>ZfUIGvkzs6e</u>	192.150.11.111	36296	④
98.114.205.102	ShAdDCFaRf	763 14	850 12	
2009-04-19T23:28:34-0400	y05ArTaWpL7	98.114.205.102	2152	⑤
192.150.11.111	ShADaFf	165088 159	4488 112	
2009-04-19T23:28:30-0400	hd4RiDVCIRI	98.114.205.102	1828	③
192.150.11.111	tcp	Signatures::Sensitive_Signature		
192.150.11.111	Isass activity			
2009-04-19T23:28:34-0400	<u>ZfUIGvkzs6e</u>	192.150.11.111	36296	④
98.114.205.102	tcp	FTP::FTP_RETR	Found FTP	
	retrieve file command			

Intrusion Detection In-Depth

We cannot get the level of detail from Bro as we do from Wireshark about the exact nature of the activity. But, remember that Wireshark requires full packet capture that may not be available or available for long. Bro, less detail available yet, it efficiently employs a compressed format for storage of logs that may allow you to retain them longer. You'll see when we cover SiLK that Bro is a compromise between the bulk of full packet capture and space saving SiLK flow that omits many of the relevant details.

Let's go through the "conn.log" to see if we can get a sense of what transpired. (1) At 23:28:28 we see that 98.114.205.102 connects to port 445 of 192.150.11.111. No data was exchanged using the history value since there is no "D" or "d" and the byte and packet count account for bytes in the IP and TCP headers only. (2) Next, 98.114.205.102 connects to 192.150.11.111 on port 1957. You can observe that some data is exchanged, but not much as reflected in 381 bytes by 98.114.205.102 and 250 by 192.150.11.111. What is interesting is that the protected network/honeypot host is now listening on port 1957.

(3) Again, 98.114.205.102 connects to port 445 of 192.150.11.111, but this time more data is exchanged and we see on the bottom of the slide that there is an entry in "signatures.log" that triggered from the Isass activity. (4) 192.150.11.111 connects to port 8884 of 98.114.205.102 and the event notice fires on FTP file retrieval. (5) Finally, 98.114.205.102 connects to port 1080 of 192.150.11.111 where 98.114.205.102 sends 165088 bytes of data.

We would have a hard time summarizing what transpired with the same granularity provided by full packet capture. Yet, we get a sense of the activity especially if the "notice.log" contains what we consider significant activity to investigate as a starting point.

Snort Plus Bro

- Snort alerts plus Bro's connection history
- Snort alerts in isolation
 - No context about what transpired before or after alert
- Bro records abbreviated data about all connections
 - Supplied event scripts don't detect malicious traffic
 - No default signatures supplied
- Together they tell a better story

Intrusion Detection In-Depth

We will discuss on Day 5 the concept of an alert-driven sensor versus a data-driven sensor. In a nutshell, Snort in NIDS mode represents an alert-driven sensor that generates alerts of notable traffic. Bro represents a data-driven sensor that can be configured to generate signature output or notices of interesting or malicious traffic. Yet it captures additional network forensic data by recording each connection. Additional analysis must be performed to discover or alert on interesting traffic.

Snort can find malicious traffic, yet does not give much context to a given alert. You know something happened; you have an artifact of the traffic characteristics, but you do not know what happened before or after. Sometimes an isolated alert gives you everything you need to know. Snort alerted on the LSASS malware, but did not have a signature to alert on the subsequent FTP download.

Bro was able to detect traffic for the signatures and events that were available or created, but there was really no indication that these were important events. We could have raised a notice that would have assigned a priority to customize our script, but not our signature. The events included with Bro do not necessarily highlight malicious traffic. As we saw, events like Bro initialization and close are informational triggers for a user script.

Yet, Bro had an outline about the connection history before and after the Snort alert traffic. In a sense, it tells a story. Combining Snort that identifies alerts in isolation, and Bro that collects a history of connections can give you a more complete story.

We concluded earlier in the course that using both tcpdump and Wireshark together enhances analysis capabilities. Similarly, using Snort and Bro together enhances detection and tracking capabilities.

Introduction to Bro

Workbook

Exercise:	"Introduction to Bro"
Introduction:	Page 35-D
Questions:	Approach #1 - Page 36-D Approach #2 - Page 40-D Extra Credit - Page 44-D
Answers:	Page 46-D

Intrusion Detection In-Depth

This page intentionally left blank.

Bro Good Reading

- Online Bro documentation
 - <http://bro.org/documentation>
- The bro "doc" directory
- NEWS file
- Man page
- Mailing list:
<http://mailman.icsi.berkeley.edu/mailman/listinfo/bro>

Intrusion Detection In-Depth

Online documentation can be found at the links mentioned above. There are some files in the "doc" directory that may be helpful as well as a "NEWS" file in the install directory telling of new features. There are man pages available and a mailing list.

Review: Open Source IDS: Snort and Bro

- Operational lifecycle provides framework for product deployment
- Detection should be the most visible and configurable phase of processing for the analyst
 - Must present a "language" to analyst to manipulate detection
 - Snort's language is its rules
 - Bro's language is its scripting
- Snort is alert/signature-driven with little known context of traffic
- Bro is event-driven with built-in traffic context via connection logs
- Bro has a policy neutral philosophy

Intrusion Detection In-Depth

First the operational lifecycle was examined to provide the foundation for the tasks necessary to run production software. We carried these steps forward to apply them to deploying Snort and/or Bro. Detection should be the most visible and configurable phase of processing for an analyst, requiring a "language" to offer the user to manipulate the detection process. Snort has a rules language while Bro has its own scripting language.

You sampled the use of two different open source IDS solutions – Snort and Bro. They both provide methods for traffic analysis, though in different ways. Snort is primarily rules-based as the means of detecting important activity via alerts that give details about the packet/stream when a rule triggers. Most of the time this is the only traffic recorded therefore a notion of context – what happened before and after the alert - is generally unknown using Snort alone.

As you've seen Bro takes a different approach to traffic analysis. It uses built-in events as entry points for the user to write scripts for the associated activity. Events themselves do not necessarily reflect malicious activity. The scripting language has the same processing capabilities as other high level languages, yet is unique in that it was created with the intent of performing traffic analysis. This makes referencing and analyzing different layers of protocols and their associated fields much more natural rather than using some retrofitted libraries to do so. And, Bro is able to give context to its analysis in the form of a connection log. It is described as a framework for network analysis, intending to provide more than just isolated notifications of activity.

Bro strives to be policy neutral. It encourages the user to assign a notion of importance to notifications. This permits a site to customize the assessment of activity in accordance with its standards.

Both Snort and Bro are remarkable open source tools that supply excellent intelligence of network activity, especially when customized properly. They both have strengths that can help you determine which is more suitable for your network. If resources permit, deploying them both provides you with alert-driven and event-driven data relating a more complete story.

Appendix of Snort Material



Intrusion Detection In-Depth

This page intentionally left blank.



Installation

Intrusion Detection In-Depth

This page intentionally left blank.

Getting Started with Snort

- Main web site for Snort is at <http://www.snort.org>
- Downloading Snort
 - Source tarball for *nix
 - Linux .rpm format
 - Win32 self-extracting executables
 - Solaris/FreeBSD/OpenBSD packages and other versions available for download but from other sites

Intrusion Detection In-Depth

To get up and running quickly with Snort, start by downloading the software from www.snort.org. There are a variety of binary packages available, including Linux .rpm format, and Windows executables with integrated installers.

If possible, the recommendation is to install from source code since that assures that you have the most current version and that you can compile optional features if you so desire. Also, it is best to download libpcap from tcpdump.org and install it from source as well if not already installed.

Preparing to Install Snort - *nix

- May have to install libpcap first
 - If you're running tcpdump, you already have it
 - Otherwise, get it from <http://www.tcpdump.org>
- Snort versions 2.9+ must install Data Acquisition(DAQ) library
 - Requires libdnet to be installed:
 - Download daq from:
<https://www.snort.org/downloads>

Intrusion Detection In-Depth

The Data Acquisition Library (DAQ) essentially detaches Snort from the software that acquires the packets. This permits the use of any acquisition software, although libpcap is the most common and default for DAQ.

Libpcap is a system-independent portable framework for sniffing network traffic. So you may need to get it and install it if it isn't already installed and you decide to stick with the default library. Most operating systems come with libpcap already installed. You will also need gcc to compile both libpcap and Snort.

DAQ requires libdnet that facilitates low-level networking operations to be installed. The most current version can be found at:

<http://code.google.com/p/libdnet/downloads/detail?name=libdnet-1.12.tgz&can=2&q=>

Once the prerequisite software is in place, get and install the DAQ from:

<https://www.snort.org/downloads>

Unpacking and Building Snort - *nix

- Create a directory to store Snort files
 - Recommendation is that it not have the Snort version number in it like `/usr/local/etc/snort`
- Grab the Snort tarball from www.snort.org
- Unpack the tarball as normal:

```
tar -zxvf snort-2.##.##.tar.gz
```

- Many build-time options available
- Build phase

```
./configure; make; sudo make install
```

Intrusion Detection In-Depth

First create a directory to store all files associated with Snort. The recommendation is for the directory name to be version neutral, something like `/usr/local/etc/snort`. The reason for this is that it makes use of the automated rules updating program Puled Pork simplr. Otherwise, you might end up with a top directory with a Snort version number that doesn't match the version number of the code and rules of its subdirectories. It might be easiest to create a directory of `/usr/local/etc`, place the version download in there and rename it to "snort". Here is how.

Fetch the download file from the Snort site <http://www.snort.org> and place it in your new directory `/usr/local/etc`. There is a gzip'ed tarball named something similar to `snort-2.##.##.tar.gz`. The `##.##` should be replaced by the latest version of Snort 2. Next, execute `tar -zxvf snort-2.##.##.tar.gz`. It will create a `snort-2.##.##` directory and load all the contents into it. Rename that directory to "snort" to get a more generic name for future upgrades. At this point, change directories to the snort directory.

You might discover that you need to change the default configuration if you wish to enable some of Snort's features such as debugging and performance profiling or monitoring. All build-time options are available by running the `./configure -help` option. As well, there is a file named `INSTALL` in the `doc` directory that describes build-time options.

Once you've decided on the build-time options prepare the installation by running `./configure` with selected build-time options. Next compile the code with the `make` command. And, finally, install Snort in system directories with `sudo make install`. If all goes well, Snort is fully installed and ready to run.

Snort Rule Distribution

- Snort rules are downloaded separately
- Different rule sets to consider:
 - Subscription: Paid service offering VRT rules
 - Registered: VRT rules, 30 days late
 - Community: Rules written by community
 - Emerging Threats: Commercial and open source rules

Intrusion Detection In-Depth

One important point to remember is that Snort rules are distributed separately from Snort code. There are several ways to stay current on rules after you download your initial set of rules. A set of rules known as the Vulnerability Research Team (VRT) certified subscription rules is written by Sourcefire/Cisco employees after learning of threats, creating applicable rules, testing those rules for effectiveness and efficiency, and finally releasing them. These are high quality rules from researchers who do this type of work for a living. These are available to paid subscribers upon release. The registered rule set is the same as the subscription rule set, available at no cost to users who sign up for the rule set, but 30 days after the subscription users receive those rules. You'll need to sign up for a Snort account at http://www.snort.org/users/sign_up to access these rules.

Another option is the community rule set that requires no subscription, is free, and available to anyone at the time of release. These rules are released on a daily basis under the auspices of Sourcefire/Cisco at the same site offering the subscription and registered rules. Users can submit rules and get credit for their submissions to this rule set. In addition, VRT examines these rules to make sure that they conform to VRT rule conventions and do not contain any obvious egregious flaws such as those that will be a big burden on processing. However, the accuracy of these rules is not validated. This community rule set must be downloaded separately from the licensed rule set.

The community rule set is reminiscent of the early days of Snort when user-contributed rules were common and encouraged. This stopped when Sourcefire began the subscription/licensed versions of the rule set. They understood that the rules had to be of the highest quality to be offered to the paying subscribers. User-contributed rules were, at times, inaccurate and inefficient. But, it appears after a multi-year hiatus the concept of user-contributed rules is back.

In the years after Snort went to the subscription-based releases, the active user community felt alienated. An outside group, started as bleedingsnort.org, now Emerging Threats was formed to be an alternate public repository for community rules. They now too have a commercial and open source offering of rules.

Currently, snort.org manages the rule distribution from repositories. It should be noted that it's not good practice to let your rule sets get stale. The VRT rule sets have end of life policies where older versions will no longer be available or supported.

Installing Rules

- Download rule sets
- Uncompress the rule set in the parent Snort directory:

```
/usr/local/etc/snort# tar -zxvf snortrules-snapshot-####.tar.gz
```

Intrusion Detection In-Depth

Download Snort rule sets from the current website. The repository has changed locations many times recently and may change again due to the Cisco acquisition. So rather than give out misinformation, it is probably best to do an Internet search for "snort rules" until the Cisco purchase settles and websites converge or remain the same.

Move these rule sets into the parent Snort directory and uncompress the package. For instance, suppose you decided to house Snort's files in "/usr/local/etc/snort" – the same directory as your code and configuration files.

```
/usr/local/etc/snort# tar -zxvf snortrules-snapshot-####.tar.gz
```

This will uncompress the Snort rules file into a created directory called "rules" for conventional rules and another called "so_rules" for shared object rules that we will soon discuss. You are now ready to reference them in your "snort.conf" configuration file.

Shared Object Rules

- Precompiled binary rules
 - Detect exploits with binary code
 - Obfuscate the source code for the rule
 - In order to provide coverage for zero day exploits that Sourcefire/Cisco learns
 - Done to protect confidential IP

Intrusion Detection In-Depth

Snort has been using shared object rules since the 2.6 version to detect a multitude of attacks. These precompiled binary rules provide protection, say for instance against a zero day attack, without overtly publishing the details of these vulnerabilities. There are also times when Sourcefire/Cisco has a contract with a research company or vendor, such as Microsoft, to learn of new vulnerabilities. An agreement permits Sourcefire/Cisco to write a detection rule for a vulnerability, but not expose the details of the 3rd party's intellectual property.

A good example of when shared object rules are used is when a zero-day attack exists with no known patch. A regular rule might give enough insight about the attack that other skilled individuals might be able to determine the nature of the vulnerability and perhaps write more or better exploits.

S.O. Rules and What They Mean to You....

- Additional installation steps:
 - Create a system directory to house dynamic rules/shared object modules

```
mkdir /usr/local/lib/snort_dynamicrules
```
 - Copy the appropriate precompiled modules for your OS/architecture/Snort version to the system directory

```
cp mysnortdir/so_rules/precompiled/  
Ubuntu-10-4/i386/2.##.##*.so  
/usr/local/lib/snort_dynamicrules/
```
- Snort currently ships with precompiled rules for:
 - Various Linux distros
 - BSD's

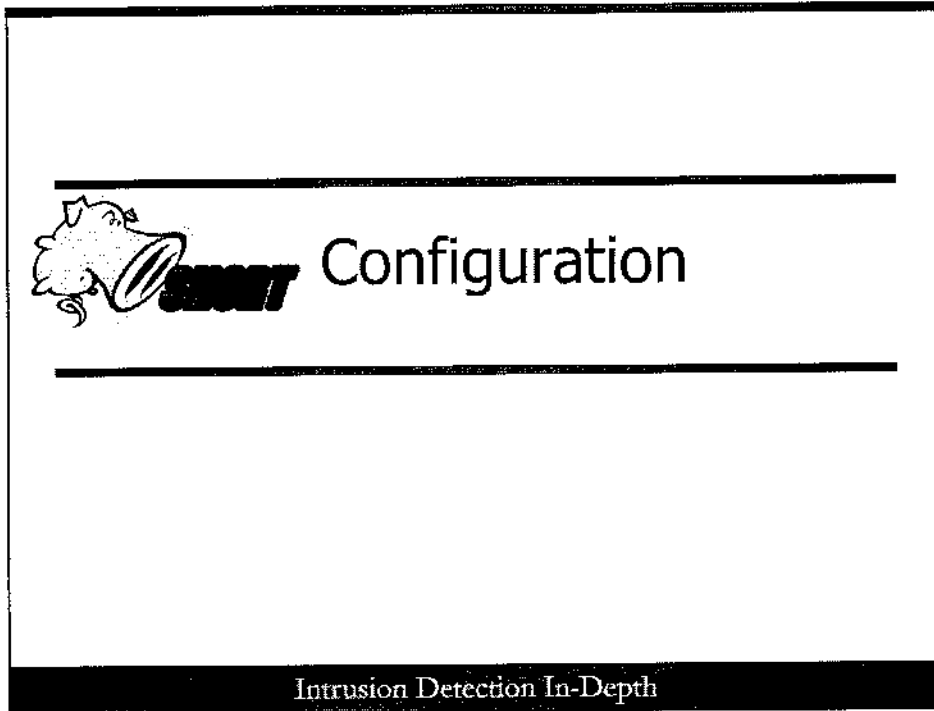
Intrusion Detection In-Depth

As previously mentioned, one of the first things to do after you have compiled and installed Snort is to go get the rules tarball and uncompress it in your Snort install directory. This will create a "rules" parent directory with regular rules subdirectories as well as a parent "so_rules" directory with the shared object rules and associated executable for each rule.

If you take a look around at the "so_rules" directory you'll see several files that end in *rules containing the actual shared object rules. We will learn more about conventional rules and their syntax and keywords in an upcoming section. If you look at a shared object rule, it looks different than a regular rule. Specifically, it has a metadata tag of "engine shared" and a "soid" – a shared object number identifier. These rules call code from the shared object library directory that you'll create. By default, Snort expects to find the shared object rules code in the system directory "/usr/local/lib/snort_dynamicrules"; make sure it exists or configure the "snort.conf" file to designate a different location should you decide to create and use a different one.

The subdirectory named "precompiled" under the "so_rules" directory contains the actual binary code for the SO rules. Navigating to that directory, you will discover the list of supported operating systems precompiled modules. There may be several subdirectories beneath each with the system architecture and Snort version. For instance, say you'd like to use the precompiled shared object rules for Ubuntu-10.4. Change directories to "Ubuntu-10-4" and you'll find support for 32 and 64-bit architectures. Let's say we're using a 32-bit machine so we select "i386" and find support for various Snort versions. We're using version 2.##.##. If you look in this directory, you'll find various modules ending in ".so". Make sure that this version number is the same as the Snort release number you just installed, otherwise you'll end up with an error and Snort will not run. If you don't find precompiled rules for your OS, it is possible others may work.

The final step is to copy all these modules to the system directory where the Snort configuration file expects to find them. We use the default of "/usr/local/lib/snort_dynamicrules".



This page intentionally left blank.

Configuring Snort As an IDS

Possible snort.conf configuration changes:

1. Set the network variables
2. Configure the decoders
3. Configure the base detection engine
4. Configure dynamic loaded libraries
5. Configure preprocessors
6. Configure output plug-ins
7. Customize your rule set
8. Customize preprocessor and decoder rule set
9. Customize shared object rule set

Intrusion Detection In-Depth

This section will cover in-depth configuration of Snort and the "snort.conf" file for use as a network IDS or IPS. The configuration file is the main mechanism to define and modify many options available in Snort.

Examine the "snort.conf" file. It lists nine tasks to perform to customize Snort for your network. It is possible to use the default configuration for all the steps; you don't need to perform any configuration to get Snort to run if it is properly installed. However, you may end up with a configuration that is not suited to the needs of your site. The "snort.conf" file is fairly liberal about its configuration, meaning that it might include preprocessors and rules that you do not need. This can lead to more meaningless alerts.

So, it is a good idea to peruse "snort.conf", read the comments, and make some changes for your network. This can be an iterative refined process of making minor initial changes, such as defining the protected network IP addresses, running Snort, and tuning the configuration as you become more familiar with Snort and your network traffic.

Another good practice is to record any changes you make in "snort.conf". Perhaps you have some software that facilitates it. At a minimum record the changes in some file or better yet, make comments in the "snort.conf" file itself. Make sure you maintain and copy any comments you make to upgraded versions of Snort with a new "snort.conf". Before you begin making any changes, it might be helpful to copy the "snort.conf" file to a renamed one in the same directory in case something goes horribly wrong.

Step 1: Set the Network Variables (1)

Format:

```
var <variable_name> <value>
portvar <variable_name> <[ports]>
ipvar <variable_name> <[IPs]>
```

To reference, place '\$' in front of variable name

```
ipvar HOME_NET 192.168.5.0/24

alert tcp any any -> $HOME_NET 6666 (msg: "IRC
connection"; sid: 1111111;)
```

If variable is not defined, but called from a rule, Snort will die at run time

Intrusion Detection In-Depth

The use of variables allows you to make simple changes that affect an entire rule set, for instance. Let's take the example of defining a HOME_NET value. The HOME_NET variable is a special one that identifies the protected network(s). It is assigned a default value of "any" in "snort.conf", meaning that any and all networks are considered protected. This value is supplied as the default so that Snort can run with no user alterations to this or any other value. Later, hopefully, when a user becomes more familiar with Snort, more appropriate values can be set.

A variable is defined in "snort.conf" by using the variable name followed by its value(s). For instance, suppose you wanted to assign the variable SSH_PORTS the values of 22 and 2222. This is accomplished with "portvar SSH_PORTS [22, 2222]". When a variable is referenced, it requires a dollar sign in front of it as \$HOME_NET shown in the slide.

The variables "portvar" and "ipvar" have special meaning and reference port numbers and IP addresses respectively. Other variables that are defined are prefaced with the keyword "var". You must define the location of the rules in "snort.conf" such as "var RULE_PATH ../rules" indicating that the rules are located in a directory named "rules" above the location of "snort.conf".

You must have a variable definition prior to any variable being referenced in a configuration or rule file. If you don't, the Snort process will die at run time.

Step 1: Set the Network Variables (2)

Set protected network value to 10.1.1.0/24

Specify at the command line with the -S option:

```
snort -S HOME_NET=10.1.1.0/24
```

Specify in snort.conf

```
ipvar HOME_NET 10.1.1.0/24
ipvar HOME_NET [10.1.1.0/24,192.168.0.0/16]
ipvar EXTERNAL_NET !$HOME_NET
```

Intrusion Detection In-Depth

You define a variable in "snort.conf" (usually at the top of the file) and the variable value will be substituted wherever referenced in the configuration file and any included rule files. A second option is to assign variable values on the command line. This avoids hard-coding the value so it is more flexible, but you have to remember to always assign a value if this is the option you chose, whereas "snort.conf" needs to be configured a single time. If you assign both a command line value and a configuration/rule set value to the same variable, the variable defined on the command line value is used. The command line option of setting variable values is mostly used for testing or debugging purposes.

If you were to use the command line option of assigning HOME_NET a value, you would use the format of "-S HOME_NET=10.1.1.0/24" where the variable and value are separated by the equal sign with no intervening spaces. This is not the recommended place to configure the value, however it makes more sense to employ if you are doing some kind of testing and don't want to alter "snort.conf".

Setting HOME_NET in "snort.conf" is accomplished by placing the following entry in "snort.conf" "ipvar HOME_NET 10.1.1.0/24" where ipvar is a reserved word that represents one or more IP addresses. A comma separates values when defining multiple values and right and left brackets surround the values. There are variables other than HOME_NET in the "snort.conf" file that have ipvar settings, such as DNS_SERVERS and SMTP_SERVERS.

Another important ipvar variable setting is EXTERNAL_NET. It represents any network that is not the HOME_NET – in other words, any network that is not protected. It is typically set in "snort.conf" with a value of "ipvar EXTERNAL_NET !\$HOME_NET". The "!" or bang indicates "not" so EXTERNAL_NET is any value that is not in the HOME_NET setting to designate all other networks other than your protected one(s).

Step 2: Configure the Decoders

- Decoders parse packets to discover underlying layers and protocols
- Also looks for some anomalies in the protocols
- See doc/README.(decoder name), i.e. README.stream5
 - Describes what a decoder is
 - How to configure a decoder
 - Lists all decoders that can be used
- Examples:
 - config enable_decode_oversized_alerts
 - config disable_tcpopt_alerts

Intrusion Detection In-Depth

Decoders parse packets as they are ingested and break them apart by their underlying protocols such as IP, TCP, etc. and analyze them. They examine the packet according to protocol specifications to determine if there are any anomalies discovered. They can potentially alert on these anomalies, if so configured.

The best place to learn more about decoders is to look at the "doc" subdirectory in your "snort" directory. It describes in more detail what a decoder is, how to configure a decoder, lists all the decoders available, and gives Snort ID numbers for each of the decoder options. Speaking of the "doc" directory – take a look at all the README files found in it. They are an excellent source of information that may not be found elsewhere.

Some decoders come enabled by default and can be disabled in "snort.conf". Conversely, some come disabled and may be enabled. You may also find decoders that are commented out that either enable or disable a particular decoder if uncommented. Take for example enabling alerting for packets that contain length fields that are greater than the actual number of bytes that follow – like an IP datagram that doesn't contain the number of bytes indicated in the IP header datagram length. This decoder comes enabled with "config enable_decode_oversized_alerts". Anomalies with TCP options are enabled by default and can be disabled with "config disable_tcpopt_alerts".

This is where to indicate that you want Snort to run as an IPS. Inline mode can be specified with the directive "config daq_mode: inline". A DAQ "type" must also be configured; options for inline mode include "afpacket", "nfq", "ipfw", and "dump". See the Snort User's Manual for a more comprehensive description of these types.

Steps 3 and 4: Configure Base Detection Engine and Dynamic Load Libraries

- Customize configuration values found in `snort.conf` under "Step 3" for post packet and preprocessor inspection

```
config profile_rules: print 10, sort avg_ticks
```

- Indicate directory locations associated with dynamic processing for shared object rules in "Step 4"

- Code for SO preprocessing
- Code for SO engine
- Code for SO rules

```
dynamicpreprocessor directory \
/usr/local/lib/snort_dynamicpreprocessor/
```

Intrusion Detection In-Depth

We've combined steps 3 and 4 in this slide since they require minimal configuration. The base detection engine configurations apply to processing after passing through packet and preprocessor phases, but before the rules detection phase. These decoders define some parameter values for rule detection such as the search algorithm to be used in parsing payload and performance monitoring behavior when enabled. The configuration options for the base detection engine are listed in "snort.conf" following the comment "Step #3". The comment says that information can be found in README.decode, however the values listed in "snort.conf" are not found there, but can be found in other README files in the same directory. Grep for a particular configuration option name in the snort "doc" subdirectory to find the specific README file.

The sample base detection engine "config profile_rules: print 10, sort avg_ticks" enables rule performance monitoring, printing the top 10 rules based on the highest average time. This will display the top 10 worst performing standard Snort rules. This is helpful when you have performance problems and would like to see one or more rules that might be responsible for the problem.

There are three directories associated with the shared object rules under "Step #4". The first contains files for preprocessing associated with SO rules in general, the second is the actual SO engine code files, and the third contains code for each of the individual SO rules.

For instance on a Linux install, the default directories created when you install Snort are:

```
/usr/local/lib/snort_dynamicpreprocessor
```

```
/usr/local/lib/snort_dynamicengine
```

```
/usr/local/lib/snort_dynamicrules
```

The first two directories are populated when you install Snort. The rules related directory is populated with the appropriate *.so files that you copied during the rules installation process.

Step 5: Configure Preprocessors

- Preprocessors allow Snort to analyze and manipulate packets in many ways
 - Traffic normalization, IP fragmentation reassembly, TCP stream reassembly, etc.
 - Preprocessors execute in the order that they're referenced in the file
 - This means protocol level preprocessors should be specified before application level ones
 - Preprocessor argument formats are unique to each one; read the comments in the snort.conf file for the appropriate syntax

Intrusion Detection In-Depth

Preprocessors allow Snort to examine and manipulate network traffic data in many ways. Preprocessors that ship with Snort can do a variety of tasks such as IP defragmentation, portscan detection, or web traffic normalization. These preprocessors are different than the dynamic preprocessors that apply only to shared object rules and code.

Preprocessors are run in the same order that they appear in the configuration file, so that means that they should be run in order of the network layer at which they operate. For instance, if an IP fragment shows up, it should be handed to the defragmenter before it is handed to the TCP stream reassembler, because the packet could potentially be defragmented into a TCP packet that needs to be examined by the stream reassembler. Snort hands fully decoded packets off to the preprocessor system before forwarding them to the detection engine.

There are comments in "snort.conf" that give you an idea of the function of a given preprocessor and its configuration options. The "doc/README" files contain some additional valuable information on particular preprocessors.

Preprocessor Configuration Example

Format:

```
preprocessor <name>:<arguments>
```

```
preprocessor stream5_global: max_tcp 8192, track_tcp  
yes, track_udp yes, track_icmp no
```

stream5 tracks 8192 concurrent TCP sessions, tracks TCP and UDP,
but not ICMP

Intrusion Detection In-Depth

Let's examine how to configure a particular preprocessor. Preprocessors are compiled into Snort at build-time and are inactive in the code until turned on at run time with a preprocessor directive in the "snort.conf" file. These directives activate and configure the preprocessors with run-time arguments unique to each individual preprocessor.

Preprocessor configuration is pretty basic. The preprocessor directive instructs the configuration system in Snort to search its available list of preprocessors for the name that follows the directive. If that name is found, the preprocessor's initialization function is called and handed the argument list that follows the colon. At this point the preprocessor is configured and is added to the preprocessor execution list inside Snort.

The stream5_global preprocessor establishes the parameters associated with stream reassembly for all traffic. These parameters and values define a maximum of 8192 concurrent sessions, and statefully track TCP and UDP, but not ICMP.

When a particular preprocessor is compiled into Snort but not activated there is no CPU overhead at run-time for leaving this preprocessor deactivated; it can be considered to be dead code.

Step 6: Configure Output Plug-ins

- Specifies alert and logging plug-ins to be used
- Several may be specified
- Options found in `snort.conf`:
 - `alert_unified2`
 - `log_unified2`
 - `unified`
 - `syslog`

```
output alert_unified2: filename snort.alerts, limit 256
```

```
output log_unified2: filename snort.logs, limit 256
```

Creates 2 binary data spools on the drive

Rotates them when they hit 256 megabytes

Intrusion Detection In-Depth

Snort used to try to process the output it generated, say for instance, by writing it a database. This further burdened a Snort sensor with an additional job of dealing with output issues as well. Eventually, the output process was separated from Snort's primary functions of sniffing data, parsing it, and alerting on it. The output plug-ins direct Snort what to do with alerts and log data.

A process known as "unified2" writes alerts, packets, and optional data to disk in binary format. Binary format takes up less space than ASCII. The unified2 process creates a standard format for the output file. You may be wondering what you do with this unique unified2 formatted file. There are programs such as Barnyard2 that ingest and parse the data to prepare it for further processing, such as storing it in a database. Unified2 includes three different operation types: alert logging, packet logging, or both. The `alert_unified2` output plug-in writes alerts to the unified2 file defined in the configuration. The `log_unified2` captures the entire packet associated with an alert and writes it to the unified2 file. And, unified2 alone performs logging both packets and alerts.

In our example, we configure the output of our alerts and logs to unified2 format. We name the files and rotate the logs every 256 MB. Each unified2 file has a name that includes a timestamp reflecting the time when logging began.

Step 7: Customize Your Rule Set (1)

- Select rules for protocols supported on the network
- Create you own rules in local.rules
- Use include statements in snort.conf

Intrusion Detection In-Depth

The "snort.conf" file uses the word "include" to indicate which of the many rule set files to include. Some of the include statements are commented out to give you the option of including them only if you determine you need them. You need to examine the rule files that are included as well as those commented out. The file name indicates the category of rules found in the given file. Examine these selections in context of the type of traffic that you run on your network. For instance, there is a "web-iis.rules" file for Microsoft's Internet Information Services (IIS) web server that includes, as you would expect, all the rules associated with IIS. If you run IIS, make sure they are included - otherwise make sure they are commented out.

As well, if you would like to write your own rules, the "local.rules" file is considered the best location for them. When rules are updated using software such as PuledPork, the local.rules file is left untouched. If instead you add a rule that you wrote to one of the Snort rule files, you run the risk of it disappearing with the next automated rules update.

Step 7: Customize Your Rule Set (2)

- RULE_PATH is a variable set in configuration file that indicates where rules are found, default value:

```
var RULE_PATH ../rules
```
- Absolute paths must be specified if the files aren't in the same directory as the referencing file
- "include" can be placed in any configuration/rule file
- Format as follows:

```
include <filename>
```

- Examples:

```
include $RULE_PATH/web-cgi.rules
include /etc/snort/customized/rpc.rules
```

Intrusion Detection In-Depth

The "snort.conf" file sets a default rules directory called "RULE_PATH" with a relative value of "../rules". This is relative to the configuration file – in this case wherever the "snort.conf" file is located, by default in Snort's "etc" subdirectory. If the rules directory cannot be defined in relative terms, or you prefer, supply an absolute name as the path directory.

The reserved word "include" simply loads the file at the point where the include is declared. If you've programmed in C, this should be a familiar concept of including the contents of the specified file at the location where it is referenced in the current file. Includes are usually used in the "snort.conf" file only to include rules files or local configuration file, but they may be used in any Snort configuration or rules file.

The format of the include is very straightforward. In the first example the default relative path found in \$RULE_PATH is used in the include for "web-cgi.rules". In the second example, an absolute path is used to identify the location of the "rpc.rules".

Step 8: Customize Preprocessor, Decoder, and Sensitive Data Rules

- Another set of rules associated with preprocessors, decoders
- Located in `preproc_rules` subdirectory in `snort` directory
 - `preprocessor.rules`
 - `decoder.rules`
 - `sensitive-data.rules`
- Detection before passed to rules engine to alert of noteworthy or anomalous packets
- Default rules directory location defined

```
var PREPROC_RULE_PATH ../preproc_rules
```

Intrusion Detection In-Depth

There is yet another rule set; it is located in the "snort" subdirectory called "preproc_rule" used for detection associated with preprocessors and decoders. This inspection and detection occur before the traditional rules processing from the main detection engine.

You will find three rules files in this directory – "decoder.rules", "preprocessor.rules", and "sensitive-data.rules". The "decoder.rules" file has rules that examine anomalies in protocols that the packet decoder parses, such as TCP, UDP and ICMP. The "preprocessor.rules" detect anomalies in any of the protocols that the preprocessor examines such as frag3 or stream5. Finally, the "sensitive-data.rules" file contains a handful of rules that try to recognize patterns that might indicate the presence of social security or credit card numbers as examples. This seems to be an attempt to deal with the problem of exfiltration of sensitive data.

The "preprocessor.rules" file is included in "snort.conf", but the "decoder.rules" and "sensitive-data.rules" are commented out. You can either include all rules or you have the option of commenting out individual rules that you do not need or find annoying in the rule files themselves.

Much like the shared object rules, you will see a very different type of rule than for standard detection rules if you go to the "preproc_rules" directory and look at any of the rules files. Each one has a Snort ID and some metadata fields associated with it, yet you see no conventional rule header that defines the protocol, source and destination IPs/ports to be found in a packet. That is because these rules do not relate to a particular source or destination; they examine non-conformity to a given protocol that is parsed and examined by both the preprocessor and decoder operations or sensitive data that can be found in any packet.

Unlike the standard rules, the user cannot add rules to "preprocessor.rules", "decoder.rules", or "sensitive-data.rules". The only customization that can be done is to change the rule action from alert to any of the other available action options that are discussed when we cover conventional Snort rules. As with standard rules, the "snort.conf" contains a variable "PREPROC_RULE_PATH" that defines the directory housing the preprocessor rules. This variable is referenced in the include statements in "snort.conf" for these three types of rules.

Step 9: Customize Your Shared Object Rules

- Dynamic library rules where detection of given activity is obfuscated
- These are located in `so_rules`, with associated code in Snort dynamic libraries
- Not included by default
- Default SO rules directory name defined

```
var SO_RULE_PATH ../so_rules
```

Intrusion Detection In-Depth

After our long configuration journey, we've reached the last step in the "snort.conf" configuration tasks. Can you believe it – another type of rule configuration. Let's recap - so far we've had standard rule configuration, preprocessor and decode rules via the use of includes and now the shared object rules. We covered the theory and installation of these rules when we discussed installing rules.

The SO rules reside in the `so_rules` subdirectory of the snort directory. These rules have most of the expected fields associated with standard rules, yet the detection methods (search for content, etc.) are not present. You can think of these rules as pointing to the actual detection code in one of Snort's dynamic libraries.

The includes with the SO rule directory names are commented out by default in "snort.conf" so you will have to look at the names and types of SO rules that exist and compare the protocol inspection performed with the protocols on your network to determine which to use.

As with standard rules, the "snort.conf" contains a variable "SO_RULE_PATH" that defines the location of the SO rules directory. This variable is referenced in the include statements in "snort.conf" for the SO rules.

Reload a Snort Configuration File

- Not necessary to restart Snort when changes made to `snort.conf`
- Must be configured with `--enable-reload` at build-time

```
$ kill -SIGHUP <snort pid>
```

- There are several non-reloadable configuration options, see Snort User's Manual for the list

Intrusion Detection In-Depth

It is possible that you may want to make immediate changes to a running instance of Snort once you alter your standard Snort rule set. Or perhaps you add or delete a preprocessor and want the change to take effect immediately. You used to have to restart Snort to accomplish this with the negative consequences of missing traffic while Snort restarted. Now, you have the option to reload the Snort configuration file and its included rules as Snort is running.

Snort is able to handle this by having a different thread for processing the configuration file from the main Snort processing. When the SIGHUP signal is used to kill the current Snort process ID, the main Snort thread will swap in the new configuration without interruption of performing its duties of sniffing and processing packets. You must build Snort to do this at configure time by including the command line option of `--enable-reload`.

There are several configuration options that are not candidates for reloading. These include configuration options such as modifying shared object rules as well as some "config" options in "snort.conf" along with some options for preprocessors. Check the Snort User's Manual since there are several cases and conditions listed.

One caveat offered is to make sure you check your new configuration file with "snort -c snort.conf -T", where the "T" checks "snort.conf" file for errors. Otherwise, errors in the new configuration file will cause Snort to die.



Updating

Intrusion Detection In-Depth

This page intentionally left blank.

Updating Snort

- Snort "engine" (code) and rules are on different update cycles, rules more frequently than code
- Important to stay current and keep the two in sync
- Snort update process similar to install process
 - Examine new functionality
 - Examine new snort.conf
 - Transfer your local.rules
 - Make a copy of the old version customizations before installing new one
- Use change and configuration management to note changes made

Intrusion Detection In-Depth

You need to update Snort code periodically when new versions and rules are released. Rules are updated quite frequently whereas there may be several months between updated Snort releases. The <http://blog.snort.org> site announces new Snort code and rule releases. As with most software, it is probably best to give the latest code release some time to settle before immediately installing it as there may be issues. You can be an early user if you are familiar with Snort and are prepared for a glitch here and there, and would like to pioneer the way for others.

Snort releases eventually expire or reach the end of life after several years and are no longer supported. You are on your own if you experience problems. Some new Snort releases include new rule functionality available only if you update the Snort code. If you are running an older release of code you will not be able to take advantage of the new rule functionality. Worse yet, is that you update a set of rules with the new functionality that breaks Snort because it cannot deal with the new rule syntax. So, it is important to stay current with both the code and rules.

Upgrading to a new release is very similar to initial installation of Snort. Read the file "RELEASE.NOTES" in the install directory to learn of the changes in the new release and any potential configuration changes that you must make. There may be new options in the "snort.conf" file so it is important to use the new "snort.conf" and change it with the customizations (variables settings, preprocessor selections, and rule inclusions, to name a few) you made to the previous version "snort.conf" file.

You may elect to install the new version in an entirely new directory or overwrite an existing one. If you do the latter, make sure you create a backup of the existing directory to keep your customizations and have the ability to revert to the previous version in case something goes wrong with the upgrade. You will need to transfer any other customizations like your "local.rules" file to the new version. As with other system changes of any type, make sure that you record your changes via change and configuration management tools. This documents the changes for others and can be referenced in case of issues.

Updating Rules

- Snort-issued rules have an End of Life policy
- Options for updating:
 - Manual
 - PulledPork
 - Oinkmaster (available but not well maintained)

Intrusion Detection In-Depth

Snort rules eventually have an end of life policy where old rule options may not be supported. It is essential that you update your rules before that happens. It is best to stay current if at all possible. You can manually install the new rules that you download from <https://www.snort.org/downloads/#rule-downloads>. Download the new rule set and install it as described in the Installation discussion on Snort rules.

Another option is PulledPork; a perl program that can update the rules for you; we'll discuss that more on the next slide.

An older rules updating program is known as Oinkmaster available at <http://oinkmaster.sourceforge.net>. It is not being actively maintained so use it at your own risk.

If resources permit, it is prudent to have a test sensor that can be used to test the updated rules before pushing them out to the production sensor(s). This staging sensor can be used to run the new rules to discover any possible errors in rules, such as missing rule dependencies or Snort version and rule version mismatches. This, however, does not validate the efficacy of the new rules. Make sure that you transfer your local customized rules to the new rules if you use an updating package that does not.

Pulled Pork

- Perl script and configuration file that updates rules
 - <http://code.google.com/p/pulledpork/>
- Must be a registered Snort user
 - https://www.snort.org/users/sign_in
- Need oinkcode for automated rules load
 - <https://www.snort.org/login>
- Configure `etc/pulledpork.conf`
- May need to load some Perl libraries
- Look at README file for features, capabilities, and command syntax

Intrusion Detection In-Depth

Pulled Pork is a program that updates the latest set of rules for you. Download it from <http://code.google.com/p/pulledpork/>. It comes as a tarball so you will need to uncompress and untar it. It uses Perl so it is not necessary to compile it.

You will need to edit the `etc/pulledpork.conf` file, at a minimum with an oinkcode. This is a unique identification that you get when you register as a Snort user. This is free and just requires you to supply a valid e-mail address, create a username and password, and accept the VRT License Agreement that you can read during the process.

You can select which rules you want to update in the Pulled Pork configuration file. There is an option to download the most current rules snapshot available to you. This will download the standard rules along with the correct version of shared object rules. You can opt to download other rule sets such as the community, IP blacklisting, and Emerging Threat rules. There are a lot of embedded comments to help you configure the file and properly run it.

As a word of warning, a Perl SSL module is required to connect to www.snort.org via HTTPS during the update process. This module can be installed with the command:

```
perl -MCPAN -e 'install Crypt::Ssleay'
```

The "README" file is a good source of information about the capabilities, features and command syntax. Before you run Puled Pork you will have to change the `"pulledpork.pl"` Perl file permissions to be executable. Finally run it from the install directory, supplying it with the configuration file location:

```
./pulledpork.pl -c etc/pulledpork.conf
```



Additional Refinement

Intrusion Detection In-Depth

This page intentionally left blank.

Snort Performance Monitoring and Profiling

- Dump packet and general statistics
- Four configuration options to collect statistics about performance
 - Performance monitoring – perfmonitor preprocessor
 - Packet Performance Monitoring (PPM)
 - Rule profiling
 - Preprocessor profiling

Intrusion Detection In-Depth

There are five ways to generate statistics from monitoring Snort's performance. Many of these different options report on the same statistics, some in different ways. You may find that the documentation about some of the methods is not particularly thorough.

The first monitoring option is to print some general statistics about Snort's operation. The second way is to use `perfmom` or `perfmonitor`—short for performance monitoring that generates a more comprehensive set of statistics. Packet Performance Monitoring (PPM) examines packet and rule latency. Finally, there are options to generate statistics about preprocessors and rules.

Regardless of the solution that you use, you will find it easier if you focus on one issue at a time, or even a single configuration variable setting at a time. There is much complexity involved in performance and remediation of issues. The suggestion is to first run one of the more comprehensive monitoring tools such as dumping the statistics or the preprocessor `perfmonitor` to try to get an idea which component of Snort, such as rules or preprocessors, seems to be having problems. The other performance monitoring methods may help you with more detailed statistics about a discovered problem.

One recommendation is to capture a pcap of representative traffic on your network for a substantial amount of time, perhaps 30 minutes to an hour. If you feel that you are having issues at some time of day or day of week in particular, this would be a good sampling opportunity. Many of the statistics that are generated do not apply to real-time so you can use a copy of the production `snort.conf`, modify it to include the use of performance management preprocessors or configuration directives, and read the captured pcap back into Snort. This permits you to run the performance monitors without direct impact on the production Snort. And, it allows you to tweak suspected issues, rerun the same traffic against the same performance monitor and look for improvements. Running against the same traffic iteratively has the advantage of making your testing environment repeatable. You can apply any improvements in the configuration to the production environment.

Packet Statistics

Send Snort a SIGUSR1

Run:

```
# kill -1          (to list your available signals)
  1) SIGHUP  2) SIGINT  ...  10) SIGUSR1
# kill -s 10 procID (send signal #10 to Snort's process ID)
```

Packet I/O Totals:

```
Received:      85
Analyzed:      85 (100.000%)
Dropped:       0 ( 0.000%)
Filtered:      0 ( 0.000%)
Outstanding:   0 ( 0.000%)
Injected:      0
```

Intrusion Detection In-Depth

One way to generate some packet and other statistics is to send a SIGUSR1 signal to Snort to dump some statistics. The actual signal number associated with SIGUSR1 may not be the same for all operating systems. If you execute the "kill -l" command, the available signals will be listed. The signal number 10 is associated with the SIGUSR1 signal on this particular system, and is sent to the Snort's process ID.

Here is a sample of the statistics generated. This particular set of statistics shows an idle home network for a short period of time. As you can see 85 packets were received and analyzed; of those none was dropped, filtered, injected and all were processed (none outstanding) where the packets counted by hardware matched the number actually analyzed.

Note in the above output, no packets were dropped. Most sites tolerate up to 1% packet loss; although your environment may be different.

Other statistics are generated too that are not displayed. These include the percentage of packets by protocol, action (alerts, logged, passed) statistics, fragment3 and stream5 statistics, as well as some preprocessor session counts.

Unlike the other performance monitoring options, this one does not require you to make any "snort.conf" changes, avoiding a reload or restart as well.

Perfmonitor Preprocessor

- Many different options to log statistics
 - Specify elapsed time/packet count for each logging event
 - Specify output file for logging data or logging to console
 - Print copious statistics to console or comma-delimited file
 - Record flow IP statistic, calculate pattern matching statistics, stream performance, per port protocol distribution, max performance for processor speed, etc.
- Example:

```
preprocessor perfmonitor: time 300 pktcnt 10000 console
```

Intrusion Detection In-Depth

The perfmonitor preprocessor logs copious statistics about Snort's performance ... These can be logged to the console or to a file. The console option is the more readable one, however it is less useful for monitoring on a long-term basis. Logging to a file creates a comma-separated list of fields and values that can then be read and processed by some third-party software. The Snort User's Manual discusses this preprocessor and all the many values and parameters associated with it in the section titled "Decoder and Preprocessor".

The above example will generate general real-time statistics for a defined metric, either time or packet count, whichever is reached first, 300 seconds or 10000 packets. The computations are performed for each 300 seconds elapsed or 10000 packets counted, repeated for these conditions, until is Snort is stopped. These values were altered to 30 and 10 respectively to generate statistics in an environment that is not a production network. An excerpt of the console output is seen on the next slide.

Sample Perfmon Output

Snort Realtime Performance : Wed Oc	
Pkts Recv:	2103
Pkts Drop:	0
% Dropped:	0.000%
Blocked:	0
Injected:	0
Pkts Filtered TCP:	0
Pkts Filtered UDP:	0
Mbts/Sec:	0.114 (wire)
Mbts/Sec:	0.000 (mpls)
Mbts/Sec:	0.000 (ip fragmented)
Mbts/Sec:	0.000 (ip reassembled)
Mbts/Sec:	0.000 (tcp rebuilt)
Mbts/Sec:	0.083 (app layer)
Bytes/Pkt:	441 (wire)
Bytes/Pkt:	0 (mpls)

Intrusion Detection In-Depth

Here is a small sample of the types of statistics reported by perfmon. As you can see, it generates some of the same statistics as interrupting Snort with the SIGUSR1 signal. Perfmon statistics are very detailed, and as you might expect, create some overhead with the many values it must compute, track, and store for each metric interval.

It is best to run this when Snort is under your control so that you can stop it after you collect what you need. Otherwise, ironically perfmon itself becomes the performance problem.

Another option is to read a pcap of collected traffic into Snort with the perfmon preprocessor enabled in snort.conf. This has the advantage of not disrupting the production Snort, but has the disadvantage of incomplete results because all statistics, such as dropped packets, cannot be computed.

Packet Performance Monitoring (PPM)

Used to measure latency in packets or rules

snort.conf set-up

```
# Per Packet latency configuration
config ppm: max-pkt-time 250, \
  fastpath-expensive-packets, \
  pkt-log

# Per Rule latency configuration
config ppm: max-rule-time 200, \
  suspend-expensive-rules, \
  suspend-timeout 20, \
  rule-log alert
```

Output

```
Packet Performance Summary:
max packet time : 50 usecs
packet event    : 1
avg pkt time    : 0.633 usecs

Rule Performance Summary:
max rule time   : 50 usecs
rule events     : 0
avg nc-rule time : 0.2675 usecs
```

Intrusion Detection In-Depth

The PPM preprocessor measures latency in either/both packets and rules. The "snort.conf" set-up is displayed on the left and the results on the right.

The first PPM configuration enables packet latency logging via the "pkt-log" parameter; it measures packet latency using a 250 micro-second thresholding, and stops inspection of a packet if the 250 micro-second limit is reached denoted by the "fastpath-expensive-packets" option. If "fastpath-expensive-packets" is not used, a counter with the number of packets that should be fast-pathed is incremented by 1.

The second PPM configuration enables rules latency logging with the "rule-log" option; it measures rule latency using a 200 micro-second thresholding, and stops inspection of a rule if the 200-micro-second limit is reached denoted by the "suspend-expensive-rules" option. If "suspend-expensive-rules" is not used, a counter with the number of rules that should be fast-pathed is incremented by 1.

The output generated details the maximum time, number of events, and average times for packet or rule performance.

The PPM rule and processor profiling require build-time configure options in order to work.

The --enable-sourcefire option can be used to enable PPM, rule, and preprocessor monitoring. The --enable-ppm enables PPM only, and the --enable-perfprofiling enables rule and performance profiling only.

Rule Profiling: Rule Performance

```
config profile_rules:
  print [all| num], \
  sort <sort_option>, \
  [filename <filename> [append]]
```

num = all or number of rules to print

<sort_option> matches nomatches avg_ticks avg_ticks_per_match
avg_ticks_per_nomatch total_ticks

<filename> output file

[append] add to existing file instead of writing over

Intrusion Detection In-Depth

Snort profiling processes – rule and preprocessor - measure the duration of each process used for detection. If the system has other CPU intensive processes, the calculated statistics may not accurately reflect the actual time.

The "snort.conf" config option `profile_rules` generates statistics about rule performance. You can output statistics for all rules or a subset based on some supplied `sort_option` criteria. Examples and partial explanations of the rules profiling can be found in the User's Manual or in the "doc/README.PerfProfiling". The documentation is the same as the README file. Unfortunately, as you'll see if you examine either, the explanations are not thorough, especially for the sort options.

Sample Rule Profile Statistics (Worst 5)

config profile_rules: print 5, sort avg_ticks, filename rulestats

Num	SID	GID	Rev	Checks	Matches	Alerts	Microsecs	Avg Check	Avg Match	Avg Nonmatch	Disabled
1	19211	1	12	1	1	0	22	22.5	22.5	0.0	0
2	25234	1	1	2	0	0	24	12.2	0.0	12.2	0
3	23243	1	5	2	0	0	8	4.1	0.0	4.0	0
4	20465	1	13	2	1	0	7	3.6	6.7	0.6	0
5	20466	1	13	2	1	0	6	3.2	6.2	0.3	0

```
root@judy:~/snort-2.9.5.3/rules# grep 19211 *
file-identify.rules:alert tcp $HOME_NET any -> $EXTERNAL_NET
$HTTP_PORTS\
(msg:"FILE-IDENTIFY ZIP archive file download request"; \
flow:to_server,established; content:".zip"; fast_pattern:only; \
http_uri;pcre:"/\x2ezip([\?\x5c\x2f]|$)/smiU"; \
flowbits:set,file.zip;flowbits:noalert; metadata:service http;\
classtype:misc-activity;sid:19211; rev:12;)
```

Intrusion Detection In-Depth

Let's run the rule profile program to dump the 5 worst rules. To do so the line "config profile_rules: print 5, sort avg_ticks, file name rulestats" is placed in the "snort.conf" file. When Snort is started, it will place the results, in the default file "/var/log/snort". We place it in a file name of "rulestats" that will be created with a timestamp following the name. Here is a description of some of the statistics:

Microseconds = total time used to evaluate rule against traffic

Checks = number of times rule was evaluated after applying pattern matching

Matches = number of times all rule options matched (will fire most times, except for conditions like thresholding rules) high for rules that have no options

Avg Check = the average number of microseconds it takes to evaluate each packet against the rule

Avg Match = the average number of microseconds to evaluate each packet that has all options match a rule

Avg Nonmatch = the average number of microseconds to evaluate each packet when the rule doesn't match

The rule with SID 19211 is discovered to have the worse performance in terms of the average check and average match. The rule is displayed to show you how complicated it is. The bad performance is most likely associated with the complex pcre expression.

The number of checks can be minimized by creating early bail conditions; the average statistics can be optimized using the recommendations for writing efficient rules.

Preprocessor Profiling: Preprocessor Performance

```
config profile_preprocs:  
  print [all| num], \  
  sort <sort_option>, \  
  [filename <filename> [append]]
```

num = all or number of preprocessors to print

<sort_option> checks avg_ticks total_ticks

<filename> output file

[append] add to existing file instead of writing over

Intrusion Detection In-Depth

Preprocessor profiling divides various phases of detection to include packet decoding, preprocessing and normalization, detection based on the pattern matcher and rule option, and logging. Performance improves when less time is spent processing each packet. The statistics evaluate the duration of each stage compared to the other stages.

The "snort.conf" config option `profile_preprocs` generates statistics about preprocessor performance. You can output statistics for all preprocessors or a subset based on some supplied `sort_option` criteria. Examples and partial explanations of the preprocessor profiling can be found in the User's Manual or in "doc/README.PerfProfiling". The documentation is the same as the README file. Unfortunately, as you'll see if you examine either, the explanations are not thorough, especially for the sort options.

Sample Preprocessor Profile Statistics (Worst 5)

config profile_preprocs: print 5, sort total_ticks

Preprocessor Profile Statistics (worst 5)								
Num	Preprocessor	Layer	Checks	Exits	Microsecs	Avg Check	Pct of Caller	Pct of Total
1	s5	0	83	83	935	11.27	33.05	33.05
1	s5tcp	1	77	77	633	8.22	67.70	22.37
1	s5TcpState	2	74	74	415	5.61	65.54	14.66
1	s5TcpData	3	50	50	26	0.53	6.37	0.93
1	s5TcpPktInsert	4	4	4	10	2.58	38.97	0.36
2	s5TcpPAF	3	4	4	18	4.65	4.48	0.66
3	s5TcpFlush	3	1	1	12	12.18	2.93	0.43
1	s5TcpProcessRebuil	4	1	1	96	96.96	796.35	3.43
2	s5TcpBuildPacket	4	1	1	0	0.72	5.95	0.03
2	s5TcpNewSess	2	4	4	33	8.42	5.32	1.19
2	s5udp	1	6	6	52	8.82	5.66	1.87
2	decode	0	172	172	534	3.11	18.90	18.90
3	eventq	0	345	345	188	0.55	6.65	6.65
4	perfnon	0	90	90	186	2.07	6.58	6.58
5	detect	0	16	16	179	11.20	6.33	6.33
1	mpse	1	27	27	109	4.06	61.23	3.88
2	rule eval	1	1	1	8	8.13	4.54	0.29
1	rule tree eval	2	2	2	7	3.72	91.38	0.26
1	uricontent	3	1	1	0	0.83	11.11	0.03
2	flow	3	1	1	0	0.25	3.30	0.01
2	rtn eval	2	1	1	1	1.79	21.97	0.06
total	total	0	172	172	2839	16.46	0.00	0.00

Intrusion Detection In-Depth

This is a confusing set of statistics if you are not familiar with preprocessor layers. A layer 0 preprocessor is one that calls other subordinate ones, known as layer 1, that can call more subordinate ones and so forth. The layer column designates the depth that that preprocessor is in these nested calls. For instance preprocessor s5 has many subordinate preprocessors ranging from a depth of 0 through 4 until the next layer 0 preprocessor decode is called. The next layer 0 preprocessor has no subordinate preprocessors nor do the three preprocessors that follow it with layer 0 processors only.

The Num column is rank of this particular processor in order from worst (1) to less worse(5). You might think that the indentation steps in this column are misaligned text, but each indentation space represents a similar layer. For instance, the worst ranked layer 0 preprocessor is s5. It has a subordinate layer 1 preprocessor known as s5tcp that is also ranked worst, a layer 2 preprocessor s5TcpState is also among the worst. The first 5 entries representing s5 and some of its subordinate preprocessors rank the worst in the terms of total time (microseconds).

According to the white paper on performance tuning, there are some statistics that are especially revealing. A high number of checks means that a preprocessor may be needlessly checking ports. The recommended solution is to configure Snort to ignore traffic that it cannot inspect such as encrypted traffic. A high average check suggests that a preprocessor may have been removed but the associated ports were not removed from the ones that are reassembled by stream5. Here is a description of some of the documented fields:

Number (rank) indented for each layer.

Layer – specific number of preprocessors all subtasks of preprocessor

Checks - number of times preprocessor looked a packet for matching ports and application layers

Exits – number of exits, should always match checks

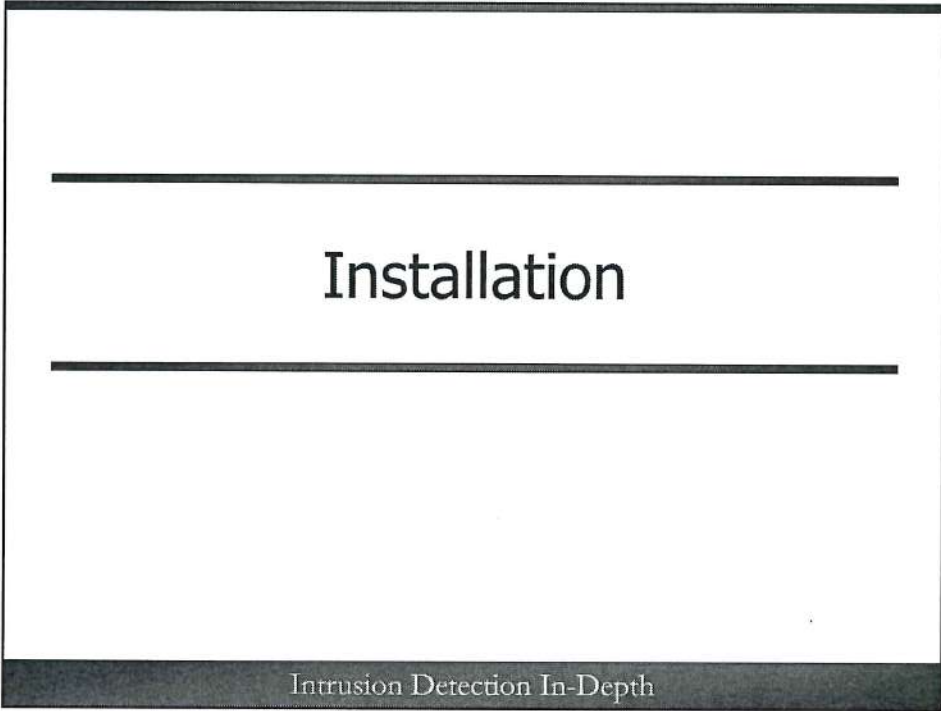
Percent of caller – what percentage of time subcomponent uses for within

Percent of total - approximate amount of CPU time that component uses

Appendix of Bro Material

Intrusion Detection In-Depth

This page intentionally left blank.



This page intentionally left blank.

Installation Details

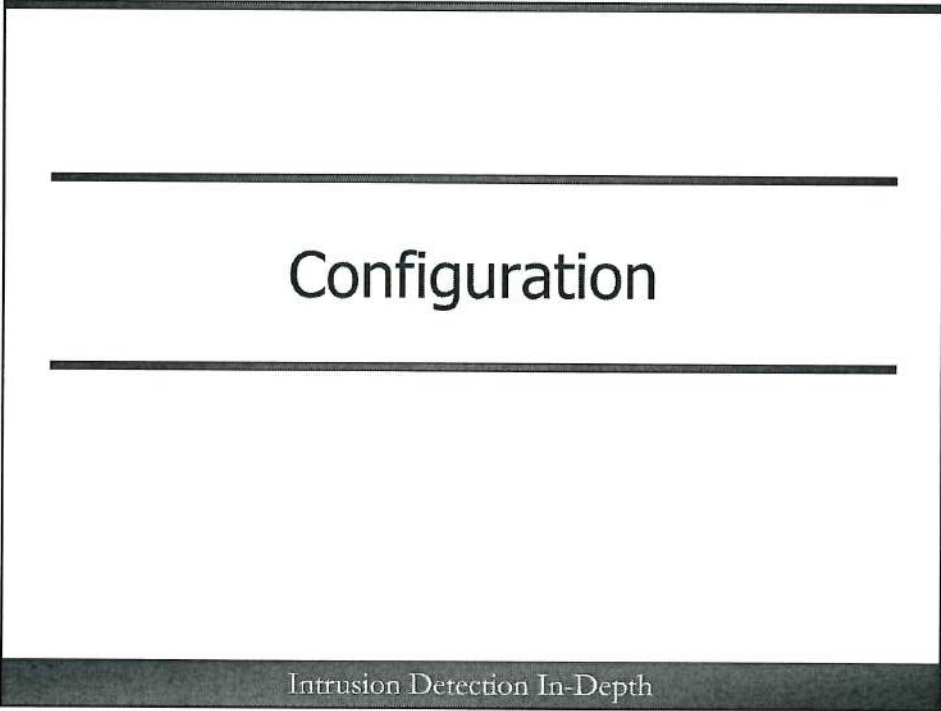
- Available in pre-built binaries in .rpm and .deb formats for Linux, .dmg for MacOS
<http://www.bro.org/download/index.html>
- Dependencies see:
<http://www.bro.org/sphinx-git/install/install.html#required-dependencies>
- Our reference for the install directory will be /usr/local/bro
- Available in source code; requires installation via:
`./configure; make; sudo make install`
- Bro needs to be installed on manager only, in both standalone and cluster mode

Intrusion Detection In-Depth

Bro downloads can be found at <http://www.bro.org/download/index.html>. Check for dependencies at <http://www.bro.org/sphinx-git/install/install.html#required-dependencies>. Bro will install its software in a default directory. For the purposes of this course, we'll use and reference the directory "/usr/local/bro" as that directory because that is the location on the VM. Other operating systems may have different default directories, but since we'll show some examples of Bro commands in a Bro environment, you will see "PREFIX" to represent the install directory.

Bro is available in binary format -- .rpm, .deb, or .dmg depending on the operating system. It can be installed from source as well, however be aware that there are several dependencies. The recommendation is to look at the link above for the necessary prerequisite software.

Installation of the Bro software is performed on the manager only for both standalone and cluster modes. We will see in the configuration section that you need only install it on the manager and Bro takes care of installing it on the remote hosts that you designate as proxy or worker(s). That is very convenient and well-conceived for expansion and flexibility.



This page intentionally left blank.

Configuration Tasks

- Configure Bro to start at system boot
- Create a cron entry to perform routine housekeeping tasks
- Make sure Bro is in your \$PATH
- Identify e-mail recipients of Bro's messages
- Change default Bro file locations in "broctl.cfg" if desired
- Identify protected network address(s)
- Identify sniffing interface(s)
- Configure Bro to run standalone or as a cluster
 - Cluster mode requires SSH public key access to worker(s)/proxy(s)
- Load up all your custom Bro scripts

Intrusion Detection In-Depth

Great; you've got Bro installed – now what? Well, as with any traffic inspection software you must configure it for your site's needs. Interaction with Bro in sniffing mode (versus reading a pcap) is performed by the "broctl" process that receives options via the command line or an interactive interface.

First, the Bro installation does not configure your system to start Bro upon system boot. You must add an entry in the appropriate location of the startup tasks to start Bro. For instance, this may be in "/etc/rc.local" on some systems. Add the entry "PREFIX/bin/broctl start", where PREFIX on the VM is "/usr/local/bro".

Next, Bro has some routine housekeeping tasks it must perform, including checking whether or not Bro is running and starting it if it is not, via the "broctl" command referenced in a cron entry. The following entry or something like it should be added to your crontab list. The entry shown here will execute every five minutes:

```
0-59/5 * * * * PREFIX/bin/broctl cron
```

Make sure that the default Bro bin directory is in your \$PATH so that you do not have to explicitly supply the absolute path each time you execute a Bro command. The exact means to perform this will be dependent upon the operating system and shell you are using.

Most of the configuration changes files are located in the "etc" subdirectory of Bro, much like Snort. The "broctl.cfg" file defines default directories used by Bro. This is also where recipients of Bro's e-mail-generated messages are assigned. The "networks.cfg" file assigns the IP addresses, with the option of expressing them in CIDR notation, of the protected network. The "node.cfg" defines the sniffing interface for a standalone implementation or IP addresses of the manager, proxy, and workers along with their interfaces when establishing a Bro cluster.

The final task is to inform Bro of the location of any customized Bro scripts and signatures that you have created. The "PREFIX/share/bro/site" directory is where customized scripts are stored. The "local.bro" file contains "@load" and "@load-sigs" statements of the location of your site's locally written scripts and signatures, much like the "include" statement used in Snort or C. There are additional files – "local-manager.bro", "local-proxy.bro", and "local-worker.bro" that load appropriate policy scripts for each of those nodes.

The subdirectory "PREFIX/share/bro/policy" contains scripts that the Bro developers consider computationally expensive or optional in nature. If you want to use specific policy scripts, you need to identify those in "local.bro" as well.

That's it for now to get you going. If you ever change any of these configuration values or add new scripts or signatures, you must inform Bro. We will see in the Running section next that this can be done in the "broctl" interactive interface by executing the commands "install" and "restart". After running Bro, you should get a better feel for how it works and you may want to change your configuration or add new scripts and signatures to extend coverage.

Special Configuration for Bro Cluster

- Cluster configuration more complicated than standalone
- Create a "bro" user
 - Need to configure any remote nodes to run SSH server
 - Need to create bro user SSH public key on manager and copy it to remote node(s)
 - Must be able to write to /usr/local/bro (install directory) on all nodes
 - Must have the privilege/capability to sniff on the worker nodes
 - Must have the same power/privileges as root/superuser

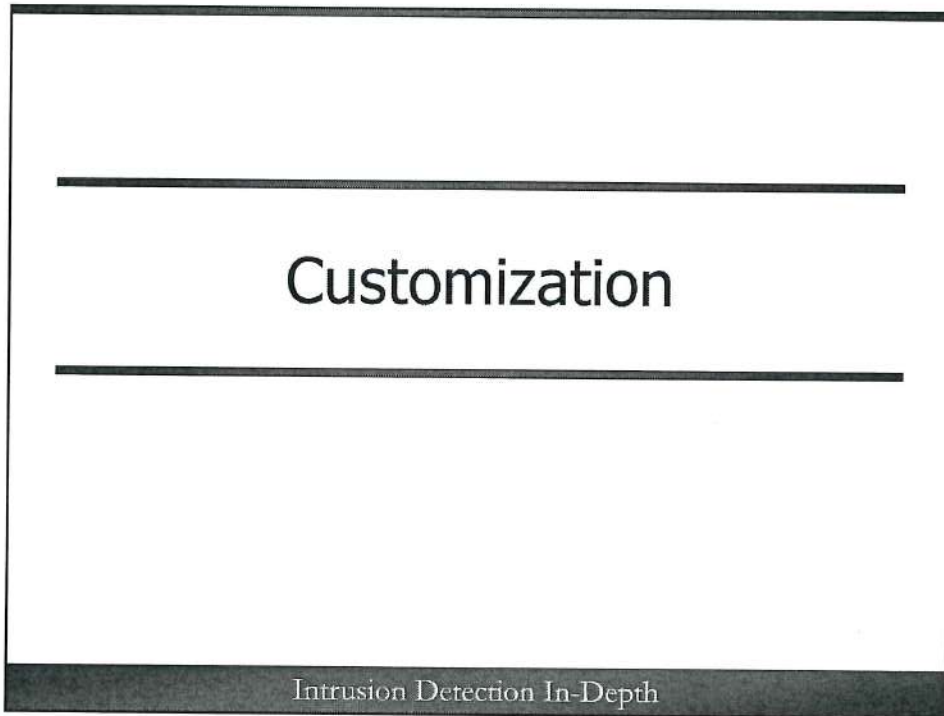
Intrusion Detection In-Depth

In standalone mode, Bro can be run as root. And, in fact, it is much simpler to run as root since creating a "bro" user requires some additional, and not necessarily simple, configuration changes. In cluster mode, the recommendation is to create a "bro" user for all the nodes. The Bro manager must be able to communicate with any remote node via SSH, meaning that some kind of SSH server software must be running on each of the remote proxy or worker nodes. One reason that the user "bro" is created is it is dangerous practice to allow remote root login via SSH. Most SSH servers are configured to deny access to the remote root user.

And, Bro requires that the communication between the manager and remote nodes be transparent such that there is no prompt for a password or passphrase, using SSH keys. The manager must generate a private and public SSH key using the "ssh-keygen" command. The public key must be copied to the remote node's file "/home/bro/.ssh/authorized_key", assuming that "/home/bro" is the directory created for user "bro".

The "bro" user must have write access to the "/usr/local/bro" or whatever your install directory is named and all subdirectories on all nodes to allow creation of files associated with software and logs. You can accomplish this by changing the owner of the directory and all contained files to "bro". The user "bro" must be able to sniff from the network interface on any of the worker nodes. You will find that the "bro" user needs to have root or superuser powers and privileges.

All configurations are created on the manager and are copied along with required software to any remote cluster node. There is no additional configuration required on the cluster node itself. You designate the remote nodes in the "PREFIX/etc/nodc.cfg" file on the manager and Bro takes care of the rest with the "broctl install".



This page intentionally left blank.

Environment

Output logs written to working directory

```
mkdir /tmp/broscripts  
cd /tmp/broscripts  
more broinit.bro
```

Step 2: Create a new working directory

```
event bro_init()  
{  
  print ("Started bro");  
}
```

Step 3: Create our Bro Script

```
root@user:/tmp/broscripts# bro -r http.pcap broinit.bro  
Started bro
```

Step 4: Run Bro Using Script



Intrusion Detection In-Depth

broinit.bro
http.pcap

The next step you must do is either use an existing directory or create an entirely new one that will contain any log files that Bro generates since it places all its log files in the current working directory. It is probably a good idea to create a new directory just for this purpose. You need write privileges in this directory. We first create and navigate to a directory we name "/tmp/broscripts".

Let's say we call our script "broinit.bro". Scripts supplied in Bro end with the extension of ".bro", however this isn't required. The script simply has the event name "bro_init" found in the file "event.bif.bro" that will cause the script to be invoked if the code path encounters it. Many scripts can have parameters passed to them, placed between the parentheses following the event name. The "bro_init" event has no parameters available to be passed. We'll talk about the format of Bro scripts on the next slide.

Next we run Bro using with the name of the Bro script to run reading in "http.pcap", although we don't need a pcap for this particular script, we include it to represent a more conventional situation. As you see, we get the output of "Started Bro".

✦ All demonstration files for Day 4 are found in directory /home/sans/demo-pcaps/Day4-demos on the VM. Use the commands found on the slides to run the scripts.

Header Signature with BPF IP Designation

```
tcpdump -r http.pcap 'ip[19] & 62 == 62' -nt -c 1

IP 173.194.73.106.80 > 192.168.11.62.19086: Flags [S.], seq 939142132, ack 11
etc.

signature sig3 {
  header ip[19] & 62 == 62
  event "sig3 Test"
}

user@host:/tmp/broscripsts: bro -r http.pcap -s sig3
cat signatures.log | bro-cut src_addr src_port dst_addr dst_port
note sig_id event_msg
173.194.73.106      80      192.168.11.62      19086   Signatures::Sensitive_Signature
sig3      173.194.73.106: sig3 Test
```



Intrusion Detection In-Depth

sig3
http.pcap

Bro has support for finding offsets in its signature "language". The syntax is similar to tcpdump BPF. It is able to match values at offsets from the beginning of IP, TCP, UDP, or ICMP. Like BPF, the fields size can span a single or multiple bytes with of size of 1, 2, or 4. And, there is support for bit masking too.

One difference between the tcpdump/libpcap and Bro BPF support is that Bro can specify a list of values that can be matched. For instance, a valid filter is:

```
header ip[19:1] == 62, 63
```

This signature examines the value in the last octet ip[19] of the destination IP address to discover if it is either 62 or 63.

The signature in the slide looks for a value of 62 using a decimal bit mask of 62 in the 19th byte offset of the IP header. You do not need to employ a mask in this situation; it was included to offer a simple example to demonstrate Bro's bit mask support.

You cannot specify a list of values when using a bit mask; you must match a single value.

Bro-Defined Notice note Variable Value

PREFIX/share/bro/base/frameworks/notice# more weird.bro

```

module Weird;
export {
  ## Generic unusual but notice-worthy weird activity.
  redef enum Notice::Type += {
    Activity,
  };
}

```

First value in \$note:
module name

\$note = Weird::Activity

Second value in \$note:
notice type

Intrusion Detection In-Depth

Let's return to the required "note" variable and value; in our example the value is "Weird::Activity". As we discussed, this represents a notice type – either one that Bro has defined or one that you define. This particular notice type is one that Bro has defined in the code found in "PREFIX/share/bro/base/frameworks/notice/weird.bro". We see an excerpt of code from "weird.bro" in the slide.

Whenever designating a value for the variable "note", the format is "module name:notice type". The value "Weird" is taken from the module name found in "weird.bro" – as defined in the first line "module Weird". We will defer the explanation of where the value "Activity" is defined. The comment in the "weird.bro" script indicates that a notice type with the name of "Activity" is for "generic unusual but notice-worthy weird activity". Our assignment of a type of "Weird::Activity" is really not especially appropriate for raising a notice for the "bro_init" event. However, if you were to look, none of the other available Bro notice types applies to starting Bro either. Bro doesn't care if it is pertinent; it's your choice and you can use any available Bro notice types or write your own as we are about to do.

Let's revisit the value of "Activity" as the second part of the notice type. Do you see the code "redef enum Notice::Type += { Activity,};"? This is Bro's way of redefining an available notice type. In essence it is adding "Activity" as new notice type. If you've ever used Python, the "+=" is a way to add something to an existing value. In this case a new value is added to the enumerable variable "Notice::Type". Bro supports different variable types such as integer, string, etc. The "enum" type permits you to define a set of related values; in our usage we can define new "Notice::Type" values. If all of this does not make sense, don't worry. The examples that follow aim to clarify this concept.

You cannot assign any arbitrary value to the "note" variable in your NOTICE; it must be an existing Bro Notice::Type or one you define. A list of existing note values can be found at:

<http://bro.org/sphinx/bro-noticeindex.html>

As suggested, you may not find a Notice::Type that is relevant for the activity so you can write one of your own. You do not have to use a Notice::Type that fits the activity. Bro doesn't care; although it makes more sense to select an appropriate one or create a Notice::Type of your own that is pertinent to the activity. This helps someone understand the entry in "notice.log". We won't explore this topic since few students will ever write their own notice types. If you are interested in learning more or seeing how this is done, the second Extra Credit question in the Bro exercise takes you through this process.

Redefining Your Own Notice::Type

```
module TEST;
#Identify a new type
redef enum Notice::Type += {
  ABC
};
event bro_init()
{
  NOTICE([$note=TEST::ABC, $msg="My TEST"]);
}

user@host:/tmp/broscripts: bro test-ABC.bro
cat notice.log | /bro-cut note msg actions
note      msg      actions
TEST::ABC My TEST   Notice::ACTION_LOG
```

Intrusion Detection In-Depth

test-ABC.bro

Creating your own Notice::Type essentially means redefining the Notice::Type global types available to you. The Bro redef statement accomplishes this by adding to the existing Notice::Type values. We define our Notice::Type to have a value of "ABC".

In our event script customization we assign the \$note variable the value of our newly created module "TEST" with our newly created Notice::Type name of "ABC". Applying this to the "bro_init" event is nonsensical, however, since the Notice is a difficult concept and implementation to grasp, it has been simplified by a lack of involved logic. You can always experiment and create a more applicable Notice for some actual event.

Assume that this script is in "test-ABC.bro". We invoke Bro using this script; no pcap is necessary since the event chosen is the start of Bro. Some fields of the "notice.log" content are shown at the bottom of the slide, including \$note and \$msg values we assigned.

Let's go through the Process of Building a Custom Script

- Let's build a script to:
 - Search for a DNS query name of isc.sans.edu
 - Create our own Notice with a user-generated name
- Steps involved:
 - Find an appropriate event to trigger our script
 - Figure out the name of the variable that Bro uses to reference a DNS query
 - Code and run a basic test to make sure we can trigger our script and output some or all of the DNS query
 - Add our own Notice

Intrusion Detection In-Depth

Trying to learn Bro on your own is difficult and often frustrating. So, let's methodically go through the steps to demonstrate the creation of a new script. This particular script is going to search traffic for a DNS name query of isc.sans.edu and create a notice about this particular activity.

As you've learned, there are several steps involved in creating your own script and raising your own notice. Since this is all relatively new, we'll progress through this by creating several interim scripts, checking that we receive the desired results, then adding more complexity as we go until we reach our goal of adding our own notice when there is a DNS query for isc.sans.edu.

Find an Appropriate Event to Trigger Script

Entry in PREFIX/share/bro/base/protocols/dns

```
event dns_request: event(c: connection, msg: dns_msg, query: string, qtype: count, qclass: count)
```

more dns-request.bro

```
event dns_request(c: connection, msg: dns_msg, query: string, qtype: count, qclass: count)
```

```
{  
    print fmt ("DNS record=> %s", c$dns);  
}
```

user@host:/tmp/broscripts: bro -r isc.pcap dns-request.bro

```
DNS record==> [ts=1350904357.794179, uid=AIXw0p3brg1, id=[orig_h=192.168.11.62, orig_p=53091/udp, resp_h=66.35.45.7, resp_p=53/udp], proto=udp, trans_id=55740, query=isc.sans.edu, qclass=1, qclass_name=C_INTERNET, qtype=1, qtype_name=A, rcode=<uninitialized>, rcode_name=<uninitialized>, AA=F, TC=F, RD=T, RA=F, Z=0, answers=<uninitialized>, TTLS=<uninitialized>, ready=F, total_answers=<uninitialized>, total_replies=<uninitialized>]
```



Intrusion Detection In-Depth

dns-request.bro
isc.pcap

DNS is protocol-specific so we look at "PREFIX/share/bro/base/protocols/dns/main.bro" for DNS events. If you examine it, you find a likely event candidate named "dns_request" along with the parameters that are passed to it when triggered.

Our first script references the "dns_request event", passing all required parameters. We simply want to print a DNS record. The connection is known as "c" as passed to the event and DNS is known as "dns" in Bro. How do you know that "dns" is used to identify the DNS nested data structure of a connection? It was discovered by looking at "PREFIX/share/bro/base/protocols/dns/main.bro".

The results of running the script placed in file "dns-request.bro" are shown in the bottom panel. The input file isc.pcap contains a DNS request to find the IP address associated with isc.scans.edu. All the variable names and associated values found in the record are displayed. We see a variable named "query" that appears to be the field we need.

Add Some Logic

```
more dns-if.bro
event dns_request(c: connection, msg: dns_msg, query: string, qtype:
count, qclass: count)
{
  if (c$dns$query == "isc.sans.edu")
    print fmt ("Found DNS query for isc.sans.edu %s",
c$dns$query);
}
```

```
user@host:/tmp/broscripts: bro -r isc.pcap dns-if.bro
```

```
Found DNS query for isc.sans.edu isc.sans.edu
```



Let's make sure we have the field we want by adding the logic of printing a statement when the query contains `isc.sans.edu`. The variable `"cdnsquery"` is used to identify the connection and dereference the DNS portion and then the query field. When we run our script, we do indeed find that we have correctly identified the DNS query field. So, we're set to move on.

Create a Module and Custom Notice::Type

```
more dns-testmod.bro
module DNSTEST;
redef enum Notice::Type += {
    SANS_DNS
};
event dns_request(c: connection, msg: dns_msg, query: string, qtype: count,
qclass: count)
{
    if (c$dns$query == "isc.sans.edu")
        print fmt ("Found DNS query for isc.sans.edu %s",
            c$dns$query);
}

user@host:/tmp/broscripts: bro -r isc.pcap dns-testmod.bro
Found DNS query for isc.sans.edu isc.sans.edu
```



Intrusion Detection In-Depth

dns-testmod.bro
isc.pcap

Next, we create a module named DNSTEST in a file that we name "dns-testmod.bro" that contains code for the our customized Notice::Type that we call "SANS_DNS" by redefining the current values available as Notice::Type. We are not going to raise a notice to invoke this just yet; we are just preparing to raise the notice in the event "dns_request".

By adding code incrementally, especially as a novice Bro script coder, we can discover whether or not we have introduced any errors in redefining the Notice::Type and take care of them before moving on. This run is successful because the script it called prints out the message of "Found DNS query for isc.sans.edu".

Raise the Notice

```
more dns-isc.bro
-----
module DNSTEST;
redef enum Notice::Type += {
  SANS_DNS
};
event dns_request(c: connection, msg: dns_msg, query: string, qtype: count, qclass: count)
{
  if (c$dns$query == "isc.sans.edu")
    NOTICE([$note=DNSTEST::SANS_DNS,
      $msg="Found suspicious DNS query isc.sans.edu", $conn=c]);
}
-----
```

```
user@host:/tmp/broscripts:bro -r isc.pcap dns-isc.bro
```

```
Part of notice.log
```

```
192.168.11.62 53091 66.35.45.7 53 udp
DNSTEST::SANS_DNS Found suspicious DNS query isc.sans.edu 1
```



Intrusion Detection In-Depth

dns-isc.bro
isc.pcap

Finally, we are ready to raise the notice by placing the NOTICE statement in the code that gets executed when the "dns_request" event is triggered. We store the code in a file named "dns-isc.bro". The first parameter assigned is the note value that is our newly created type of "DNSTEST::SANS_DNS". Remember that this name is a combination of the containing module name and our new notice type of "SANS_DNS". We assign an appropriate message of "Found suspicious DNS query isc.sans.edu" as this is really the entire purpose of raising the notice.

We also opt to place additional information in the notice for details about the connection that triggered the event. The \$conn variable is one of several that can be defined in the NOTICE assignment. This represents the connection known as "c" as passed to the event.

A "notice.log" is created as expected. Some pertinent output is displayed from it such as the source and destination IP addresses and ports, the protocol of UDP, the notice type associated with the log entry, and our custom message.

You can create more complex logic in the script with conditions that must exist before the notice is raised. This was just meant to be a fairly simple example of how to approach raising notices and all the preceding processes that must occur before doing so.

Check if New Script Loaded, Then Cause It to Trigger

```
root@judy: cd PREFIX/logs/current
root@judy:/usr/local/bro/logs/current #
  grep "dns-isc.bro" loaded_scripts.log
  PREFIX/share/bro/site/mybro/dns-isc.bro

root@judy:/usr/local/bro/logs/current # tail -f notice.log

user:~$ nslookup isc.sans.edu

notice.log entry
192.168.11.42 59749 4.2.2.1 53 udp
DNSTEST::SANS_DNS Found suspicious DNS query isc.sans.edu
192.168.11.424.2.2.1 53 Notice::ACTION_LOG
```

Intrusion Detection In-Depth

How would you test this in production mode? Let's pick up at the point after you loaded your new script. Go to the directory "PREFIX/logs/current". See if your script was loaded by issuing a grep command for the name of your script – in this case "dns-isc.bro" in the "loaded_scripts.log".

It's there so we're ready to test the script. Specifically, we are interested in the current directory "notice.log" so we closely watch the new activity generated to make sure our script generates a notice using the command:

```
tail -f notice.log
```

Next, we issue a lookup for isc.sans.edu to try to trigger our new DNS script. And as you see, an entry is logged to "notice.log" that reflects our script output.

We're finished examining scripts, signatures, and notices that can be valuable tools for you to use in your network traffic analysis.

Inform About a New HTTP Request/Reply

```
root@user:/tmp/broscripts# more bro-http-reqresp.bro
```

```
event http_request(c: connection, method: string, original_URI: string,
    unescaped_URI: string, version: string) &priority=5
{
    print fmt ("HTTP request ==> sender: %s %s receiver: %s %s HTTP
method: %s HTTP URI: %s", c$id$orig_h, c$id$orig_p, c$id$resp_h,
c$id$resp_p, c$http$method, c$http$uri);
}
event http_reply(c: connection, version: string, code: count, reason: string)
{
    print fmt ("HTTP response ==> sender: %s %s receiver: %s %s HTTP
code: %d HTTP reason: %s", c$id$resp_h, c$id$resp_p, c$id$orig_h,
c$id$orig_p, c$http$status_code, c$http$status_msg);
}
```



Intrusion Detection In-Depth

bro-http-reqresp.bro

Next, we cover how Bro is able to parse and analyze several different application layers for use in scripting, including HTTP, as shown in the slide. The two different events above "http_request" and "http_reply" represent, as you would expect, an HTTP request and reply. You can see that there are several arguments and their respective variable types (connection, string, count) passed by the associated event. The HTTP request passes the connection, the HTTP method, the normalized URI, the pre-normalized URI, and the HTTP version. The HTTP response passes the connection, the version, the HTTP response code and reason.

Our script is triggered by an "http_request" event to perform a formatted print to dump some information – the sender and receiver IP/port pairs, the HTTP method and the HTTP URI as passed to the function. The script also supports the "http_reply" event where our code uses a formatted print of the sender and receiver IP/port pairs, along with the HTTP server's returned status code and status message.

Events can be assigned a priority attribute value ranging from -10 through 10 where higher value code is executed first. The default priority value is 0, yet the "http_request" is assigned a value of 5. The priority value is used if you have multiple scripts associated with an event so that they can be executed in an order based on the priority value.

The next slide discusses how we discovered the event name, its arguments, and the available variables for the HTTP protocol.

The Event Name, Arguments, and HTTP Variables

Event name and arguments found in \$PREFIX/share/bro/base/protocols/http/main.bro

```
event http_reply(c: connection, version: string, code: count, reason: string)
{
    print fmt ("HTTP => %s", c$http);
}
```

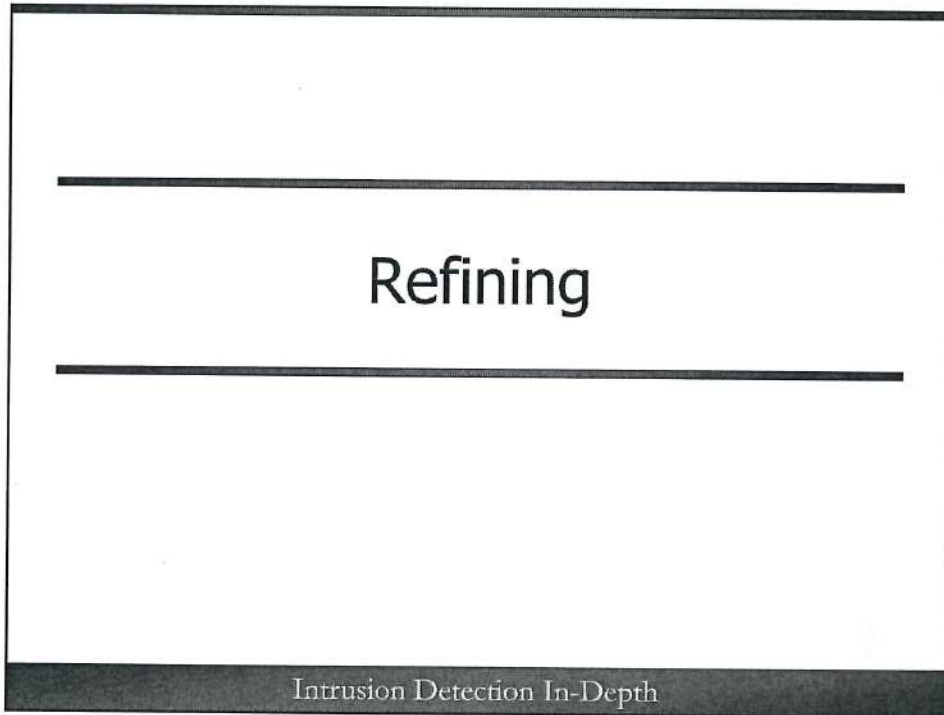
```
HTTP ==> [ts=1349511963.420526, uid=CLub3C3rLXXpoDjPx6, id=[orig_h=192.168.11.62, orig_p=49931/
tcp, resp_h=173.194.75.99, resp_p=80/tcp], trans_depth=1, method=GET, host=www.google.com,
uri=/, referrer=<uninitialized>, user_agent=Mozilla/5.0 (X11; Linux i686; rv:15.0)
Gecko/20100101 Firefox/15.0.1, request_body_len=0, response_body_len=0, status_code=302,
status_msg=Found, info_code=<uninitialized>, info_msg=<uninitialized>]
```

Intrusion Detection In-Depth

Each of the protocols supported by Bro, in this case HTTP, has a corresponding directory in "PREFIX/share/bro/base/protocols" – in this example subdirectory "http". You will find a file named "main.bro" in each of the protocol subdirectories that defines the events available to you along with the names of the passed arguments, and the variable names and descriptions of the available fields for that protocol. Let's isolate the "http_reply" event found in "PREFIX/share/bro/base/protocols/http/main.bro" to examine the variables used in our script.

Once again, you may ask how we knew argument variable names and those same fields when referenced in the script. The argument variables are those found in the "main.bro" file that has the "http_reply" event. Unfortunately, the argument names may not match the names used to reference a given field in the script itself. For instance, the "http_reply" event passes an argument called "reason", however this is known as "\$status_msg" in the script reference above.

To reveal the variable names to reference for a given protocol, print out the protocol data structure, in this instance "http". Remember that the protocol/data structure is always nested in a given connection "c" so "c\$http" lists all the HTTP variable fields and values for the current connection. This is a cumbersome way to discover the variables, yet it seems like the most informative.



This page intentionally left blank.

Output from Script

```
root@user:/tmp/broscripits# bro -r http.pcap bro-http-reqresp.bro
```

```
HTTP request ==> sender: 192.168.11.62 19086/tcp receiver: 173.194.73.106  
80/tcp HTTP method: GET HTTP URI: /
```

```
HTTP response ==> sender: 173.194.73.106 80/tcp receiver: 192.168.11.62  
19086/tcp HTTP code: 200 HTTP reason: OK
```



When the script is run reading in the file "http.pcap", we see the desired print output. Both the HTTP request and response include the IP/port pairs of the sender and receiver. The request HTTP method is GET and the HTTP URI is "/". The response returns an HTTP response code of 200 and a reason of OK meaning that the URL was found.

stats.bro

Sample output from PREFIX/logs/current/stats.log

#fields	ts	peer	mem	pkts_proc	events_proc	pkts_dropped	pkts_link	
	events_queued	lag		pkts_recv				
1377685997.949261	0	0	bro	18	0	141	8	0.000289
1377686057.949077	261	0	bro	18	261	239	235	0.000066
1377686117.949087	83	0	bro	18	83	146	146	0.000069

Intrusion Detection In-Depth

When the "stats.bro" script is loaded in the "local.bro" file, a new log named "stats.log" is created to contain the output. The output on the slide is from a home network where stats.bro was run for a couple of minutes to generate some sample log data.

Some of the more useful fields are the amount of memory currently in use, the number of processed packets, number of events – both processed and queued since the last stats interval (a default value of one minute). As well, there are the number of packets received and packets dropped and the number of packets on the link that should be the total of the received and dropped packets.

For more details about each of the fields, see the file "PREFIX/share/bro/policy/misc/stats.bro".

profiling.bro

Sample output from PREFIX/logs/current/prof.log

```
0.000000 -----
0.000000 Memory: total=17980K total_adj=0K malloced: 17850K
0.000000 Run-time: user+sys=0.0 user=0.0 sys=0.0 real=0.0
0.000000 Conns: total=0 current=0/0 ext=0 mem=0K avg=0.0 table=0K connvals=0K
0.000000 Conns: tcp=0/0 udp=0/0 icmp=0/0
0.000000 TCP-States:   Inact. Syn.  SA   Part. Est.  Fin.  Rst.
0.000000 Connections expired due to inactivity: 0
0.000000 Total reassembler data: 0K
0.000000 Timers: current=29 max=29 mem=2K lag=0.00s
0.000000 DNS_Mgr: requests=0 succesful=0 failed=0 pending=0 cached_hosts=0
           cached_addrs=0
0.000000 -----
0.000000 Threads: current=4
0.000000 packet_filter/Log::WRITER_ASCII in=1 out=0 pending=0/0 (#queue r/w: in=1/1
           out=0/0)
```

Intrusion Detection In-Depth

Loading the "profiling.bro" script immediately places the output in the file "prof.log" in the "current" log directory. It generates copious amounts of output with detailed statistics. The above is an abbreviated and edited listing of some of the statistics including memory, connections, threads, etc. If you are interested in more detail about what is generated, run the script for several minutes and examine the "prof.log".

Unlike the "stats.bro" script, "profiling.bro" has no comments in it to help understand the output.

capture-loss.bro

```
cat capture_loss.log | bro-cut -d ts ts_delta peer gaps acks  
percent_lost
```

2013-10-27T13:03:42-0400	900.000000	bro	0
23 0.000%			
2013-10-27T13:18:42-0400	900.000017	bro	1
34 2.941%			
2013-10-27T13:33:42-0400	900.000004	bro	28
151 18.543%			
2013-10-27T13:48:42-0400	900.000048	bro	1
21 4.762%			

Intrusion Detection In-Depth

Here are a few entries that were captured at six minute intervals in the "capture_loss.log" after loading "PREFIX/share/bro/policy/misc/capture-loss.bro" file. Packet loss is determined by the "number of 'gap events' (ACKs for a sequence number that's above a gap)".

In other words it appears to be examining TCP traffic only. It looks for acknowledgement numbers that are greater than the expected ones, determined by the last sequence number sent. Therefore, if there is a gap in the acknowledgement value, Bro assumes that packets have been lost since it does not see them sent with the sequence numbers that both Bro and TCP expect.

It appears that this network suffers from some significant packet loss during the third interval where Bro found 151 acknowledgements with values higher than expected, computed to be over 18% of the traffic observed. This seems somewhat suspect since this was captured on a home network that appeared to be working fine judging by responses to sent traffic. This means that Bro believes there were 151 gap acknowledgements, representing 18% of all acknowledgements from the total number of packets - about 750 sent in a duration of 6 minutes. This appears to be a problem with Bro's computations since it seems very unlikely that so many packets were dropped in such an idle environment.

For more details, see the file "PREFIX/share/bro/policy/misc/capture-loss.bro".

Execute capstats for NIC Activity

- Capstats is a tool/command executed from broctl to collect statistics of the activity on a network interface

```
[BroControl] > capstats -i nve0 -I 1
```

```
1186620936.890567 pkts=12747 kpps=12.6 kbytes=10807 mbytes=87.5  
nic_pkts=12822 nic_drops=0 u=960 t=11705 i=58 o=24 nonip=0
```

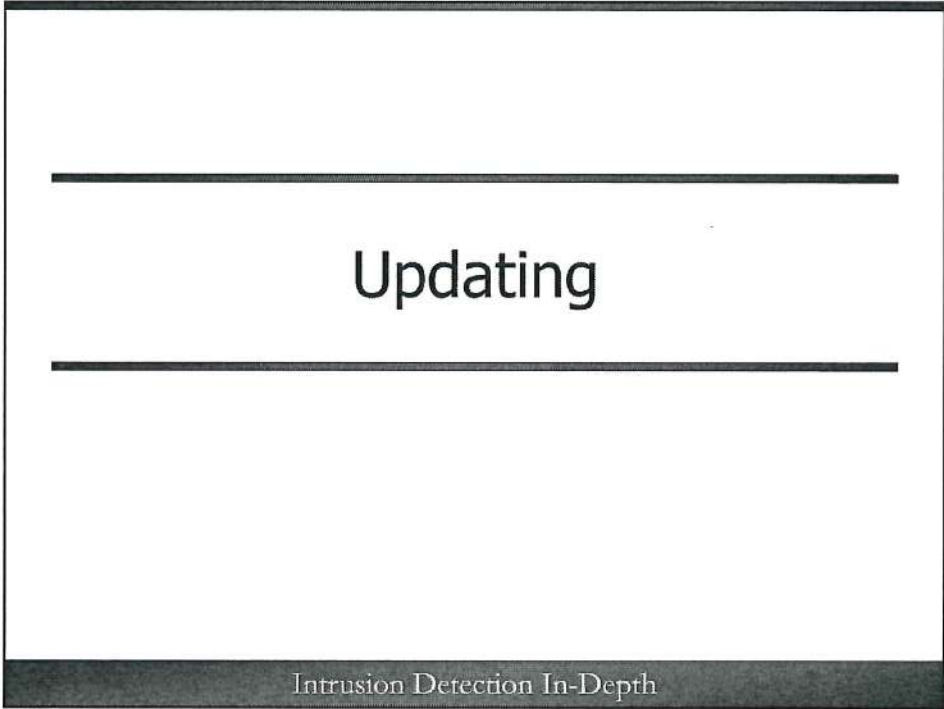
Intrusion Detection In-Depth

If you are concerned that there may be performance issues from the network interface card, the capstats command in "broctl" can assist in showing the number of packets received, the rate, and any dropped packets. The capstats command and output data were copied from the documentation to show you how it should work.

However, in trying the same command, using the same format, but using the interface of "eth0", reflecting the sniffing interface on the host where it was run, an error was received about an "unknown interface". The format of the capstats command follows:

capstats [Options] -i interface

-i --interface <interface>	Listen on interface
-d --dag	Use native DAG API
-f --filter <filter>	BPF filter
-I --interval <secs>	Stats logging interval
-l --syslog	Use syslog rather than print to stderr
-n --number <count>	Stop after outputting <number> intervals
-N --select	Use select() for live pcap (for testing only)
-p --payload <n>	Verifies that packets' payloads consist entirely of bytes of the given value.
-q --quiet <count>	Suppress output, exit code indicates >= count packets received.
-S --size <size>	Verify packets to have given <size>
-s --snaplen <size>	Use pcap snaplen <size>
-v --version	Print version and exit
-w --write <filename>	Write packets to file



This page intentionally left blank.

Updating Bro

- Software updates only
- PREFIX/share/bro/site scripts won't be overwritten
- No interim script updates
- No signatures since they are created by user only

Intrusion Detection In-Depth

Bro updates are for the software only when a new version is released. There are no regular interim script updates as yet. And, signatures are user-created so none come with Bro. You only have to ensure that whatever scripts and signatures you do create are placed in the directory supplied for your site located in "PREFIX/share/bro/site". When you upgrade, these files will stay intact. Otherwise they will be overwritten and disappear if placed in any of Bro's other directories.

Upgrade Options

- Reuse same install prefix, "site" and "etc" directories won't be overwritten
- Use a new install prefix, copy site/etc files, broctl.cfg needs to change the SpoolDir and Logdir entries
 - Revisit initial configuration tasks
 - Configure Bro to start at system boot
 - Create a cron entry to perform routine housekeeping tasks
 - Make sure Bro is in your \$PATH

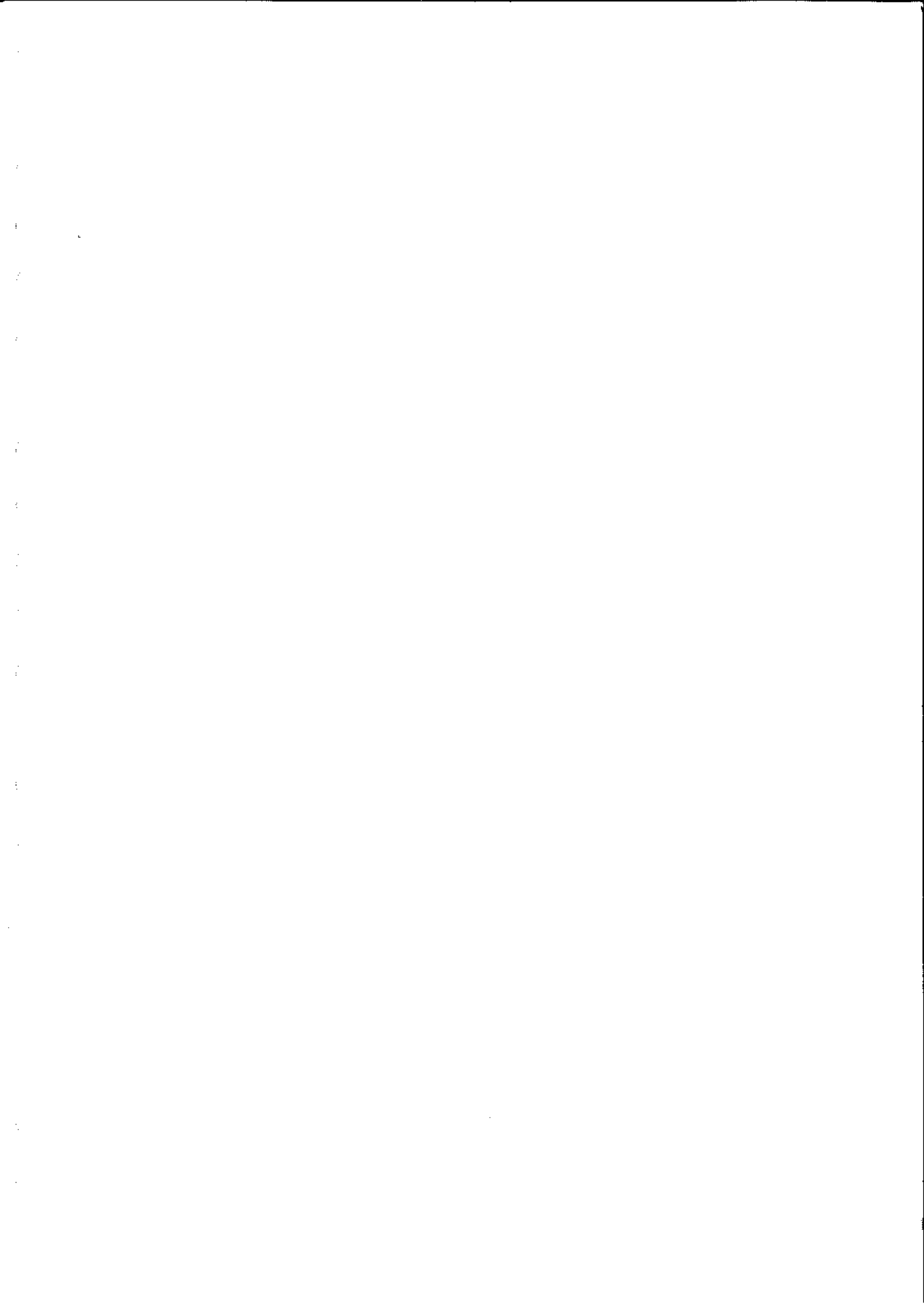
Intrusion Detection In-Depth

Bro upgrades can either use the existing Bro files for the updated software or a new base directory. The advantage of using the existing directories is that your customized files in the "site" and "etc" directories will remain untouched.

However, if you prefer to be cautious, especially when operating in production mode, it may be wise to keep the operational Bro files untouched in case you need to revert back to the previous version. You can create a new Bro base directory to install the upgraded software. Examine the files in the Bro "PREFIX/etc" directory to make configuration changes for the updated software. Also, copy any customized site-specific files from the older version "site" directory to the new version "site" directory.

Finally, revisit the configuration tasks performed upon initial installation if you've upgraded with the new base directory option. These include making sure that Bro starts at system boot with its altered location, changing the cron entry with the new directory location, and making sure that the new Bro location is in your \$PATH.

Another option is to copy your current production files in the PREFIX directory to some other location as a backup in case the Bro upgrade to PREFIX directory creates issues. If the new version works, your customized files will still be in place. If the upgrade fails, you can simply rename the backup directory to the original name.



ABOUT SANS

SANS is the most trusted and by far the largest source for information security training and certification in the world. It also develops, maintains, and makes available at no cost the largest collection of research documents about various aspects of information security, and it operates the Internet's early warning system – the Internet Storm Center. The SANS (SysAdmin, Audit, Network, Security) Institute was established in 1989 as a cooperative research and education organization. Its programs now reach more than 165,000 security professionals around the world. A range of individuals from auditors and network administrators to chief information security officers are sharing the lessons they learn and are jointly finding solutions to the challenges they face. At the heart of SANS are the many security

practitioners in varied global organizations from corporations to universities working together to help the entire information security community. SANS provides intensive, immersion training designed to help you and your staff master the practical steps necessary for defending systems and networks against the most dangerous threats – the ones being actively exploited. This training is full of important and immediately useful techniques that you can put to work as soon as you return to your office. Courses were developed through a consensus process involving hundreds of administrators, security managers, and information security professionals, and they address both security fundamentals and awareness and the in-depth technical aspects of the most crucial areas of IT security. www.sans.org

IN-DEPTH EDUCATION AND CERTIFICATION

During the past year, more than 17,000 security, networking, and system administration professionals attended multi-day, in-depth training by the world's top security practitioners and teachers. Next year, SANS programs will educate thousands more security professionals in the US and internationally.

SANS Technology Institute (STI) is the premier skills-based cybersecurity graduate school offering master's degree in information security. Our programs are hands-on and intensive, equipping students to be leaders in strengthening enterprise and global information security. Our students learn enterprise security strategies and techniques, and engage in real-world applied research, led by the top scholar-practitioners in the information security profession. Learn more about STI at www.sans.edu.

Global Information Assurance Certification (GIAC)

GIAC offer more than 25 specialized certifications in the areas of incident handling, forensics, leadership, security, penetration and audit. GIAC is ISO/ANSI/IEC 17024 accredited. The GIAC certification process validates the specific skills of security professionals with standards established on the highest benchmarks in the industry. Over 49,000 candidates have obtained GIAC certifications with hundreds more in the process. Find out more at www.giac.org.

SANS BREAKS THE NEWS

SANS NewsBites is a semi-weekly, high-level executive summary of the most important news articles that have been published on computer security during the last week. Each news item is very briefly summarized and includes a reference on the web for detailed information, if possible. www.sans.org/newsletters/newsbites

@RISK: The Consensus Security Alert is a weekly report summarizing the vulnerabilities that matter most and steps for protection. www.sans.org/newsletters/risk

Ouch! is the first consensus monthly security awareness report for end users. It shows what to look for and how to avoid phishing and other scams plus viruses and other malware using the latest attacks as examples. www.sans.org/newsletters/ouch

The Internet Storm Center (ISC) was created in 2001 following the successful detection, analysis, and widespread warning of the LiOn worm. Today, the ISC provides a free analysis and warning service to thousands of Internet users and organizations and is actively working with Internet Service Providers to fight back against the most malicious attackers. <http://isc.sans.org>

TRAINING WITHOUT TRAVEL ALTERNATIVES

Nothing beats the experience of attending a live SANS training event with incomparable instructors and guest speakers, vendor solutions expos, and myriad networking opportunities. Sometimes though, travel costs and a week away from the office are just not feasible. When limited time and/or budget keeps you or your co-workers grounded, you can still get great SANS training close to home.

SANS OnSite *Your Schedule! Lower Cost!*

With SANS OnSite program you can bring a unique combination of high-quality and world-recognized instructors to train your professionals at your location and realize significant savings.

Six reasons to consider SANS OnSite:

1. Enjoy the same great certified SANS instructors and unparalleled courseware
2. Flexible scheduling – conduct the training when it is convenient for you
3. Focus on internal security issues during class and find solutions
4. Keep staff close to home
5. Realize significant savings on travel expenses
6. Enable dispersed workforce to interact with one another in one place

DoD or DoD contractors working to meet the stringent requirements of DoD-Directive 8570? SANS OnSite is the best way to help you achieve your training and certification objectives. www.sans.org/onsite

SANS OnDemand *Online Training & Assessments – Anytime, Anywhere*

When you want access to SANS' high-quality training 'anytime, anywhere', choose our advanced online delivery method! OnDemand is designed to provide a very convenient, comprehensive, and highly effective means for information security professionals to receive the same intensive, immersion training that SANS is famous for. Students will receive:

- Up to four months of access to online training
- Hard copy of course books
- Integrated lectures by SANS top-rated instructors
- Progress reports
- Access to our SANS Virtual Mentor
- Labs and hands-on exercises
- Assessments to reinforce your knowledge throughout the course

www.sans.org/ondemand

SANS vLive *Live Virtual Training – Top SANS Instructors*

SANS vLive allows you to attend SANS courses from the convenience of your home or office! Simply log in at the scheduled times and join your instructor and classmates in an interactive virtual classroom. Classes typically meet two evenings a week for five or six weeks. No other SANS training format gives you as much time with our top instructors.

www.sans.org/vlive

SANS Simulcast *Live SANS Instruction in Multiple Locations!*

Log in to a virtual classroom to see, hear, and participate in a class as it is being presented LIVE at a SANS event! Event Simulcasts are available for many classes offered at major SANS events. We can also offer private Custom Simulcasts – perfect for organizations that need to train distributed workforces with limited travel budgets. www.sans.org/simulcast

For group programs, please contact us at groupsales@sans.org