

503.3

Signature-Based Detection

SANS

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

<https://t.me/learningnets>

PLEASE READ THE TERMS AND CONDITIONS OF THIS COURSEWARE LICENSE AGREEMENT ("CLA") CAREFULLY BEFORE USING ANY OF THE COURSEWARE ASSOCIATED WITH THE SANS COURSE. THIS IS A LEGAL AND ENFORCEABLE CONTRACT BETWEEN YOU (THE "USER") AND SANS INSTITUTE FOR THE COURSEWARE. YOU AGREE THAT THIS AGREEMENT IS ENFORCEABLE LIKE ANY WRITTEN NEGOTIATED AGREEMENT SIGNED BY YOU.

With this CLA, SANS Institute hereby grants User a personal, non-exclusive license to use the Courseware subject to the terms of this agreement. Courseware includes all printed materials, including course books and lab workbooks, as well as any digital or other media, virtual machines, and/or data sets distributed by SANS Institute to User for use in the SANS class associated with the Courseware. User agrees that the CLA is the complete and exclusive statement of agreement between SANS Institute and you and that this CLA supersedes any oral or written proposal, agreement or other communication relating to the subject matter of this CLA.

BY ACCEPTING THIS COURSEWARE, USER AGREES TO BE BOUND BY THE TERMS OF THIS CLA. BY ACCEPTING THIS SOFTWARE, USER AGREES THAT ANY BREACH OF THE TERMS OF THIS CLA MAY CAUSE IRREPARABLE HARM AND SIGNIFICANT INJURY TO SANS INSTITUTE, AND THAT SANS INSTITUTE MAY ENFORCE THESE PROVISIONS BY INJUNCTION (WITHOUT THE NECESSITY OF POSTING BOND) SPECIFIC PERFORMANCE, OR OTHER EQUITABLE RELIEF.

If User does not agree, User may return the Courseware to SANS Institute for a full refund, if applicable.

User may not copy, reproduce, re-publish, distribute, display, modify or create derivative works based upon all or any portion of the Courseware, in any medium whether printed, electronic or otherwise, for any purpose, without the express prior written consent of SANS Institute. Additionally, User may not sell, rent, lease, trade, or otherwise transfer the Courseware in any way, shape, or form without the express written consent of SANS Institute.

If any provision of this CLA is declared unenforceable in any jurisdiction, then such provision shall be deemed to be severable from this CLA and shall not affect the remainder thereof. An amendment or addendum to this CLA may accompany this Courseware.

SANS acknowledges that any and all software and/or tools, graphics, images, tables, charts or graphs presented in this Courseware are the sole property of their respective trademark/registered/copyright owners, including:

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

PMP® and PMBOK® are registered trademarks of PMI.

SOF-ELK® is a registered trademark of Lewes Technology Consulting, LLC. Used with permission.

SIFT® is a registered trademark of Harbingers, LLC. Used with permission.

Governing Law: This Agreement shall be governed by the laws of the State of Maryland, USA.



Signature-Based Detection

© 2021 David Hoelzer | All Rights Reserved | Version G0I_02

This page intentionally left blank.

Table of Contents (1)	Page
Packet Crafting for IDS/IPS	
➤ Fundamentals of Scapy and Assembling a Packet	4
➤ Writing, Reading, and Altering Packet Captures	14
<i>Packet Crafting for IDS/IPS Exercises</i>	25
Wireshark Part III	
➤ Extracting Files	26
➤ Forensic Reconstruction	36
<i>Wireshark Part III Exercises</i>	50
Application Protocols with Snort and Suricata	
➤ Introduction to Snort	51
<i>Running Snort and Snort Output Exercises</i>	65
➤ Writing Rules	66
<i>Writing Rules I Exercise</i>	77
➤ Content Options	78
➤ Streams	86
SANS	SEC503: Intrusion Detection In-Depth 2

This page intentionally left blank.

[illegible]

This page intentionally left blank.

• An immersion class in speaking packets, protocols, and hex • Ethernet • IPv4 & IPv6 • TCP, UDP, & ICMP • Wireshark & tcpdump fundamentals, filtering, & investigation	Section 2: Signature-Based Detection • Scapy for building tools and simulating signatures & behaviors • Snort & Suricata • Researching common application protocols	• Architecting for massive packet capture & analysis • Overcoming the limitations of signature-based tools • Basic Zeek usage and logs • Threat modelling • Advanced correlation using Zeek	• Guided scenarios to reinforce all skills • Elaboration on effective deployment, collection, and analysis at large scale • Moving beyond alerts to data, driven analysis, & machine learning	• Challenge questions guide/follow expert analysts' progression through real-world data
---	--	---	---	---

- Packet Crafting for IDS/IPS
- Wireshark Part III
- Application Protocols with Snort and Suricata

Signature-Based Detection

Fundamentals of Scapy and Assembling a Packet

Writing, Reading, and Altering Packet Captures

In Book 3, we really begin to make direct application into the field of intrusion detection. Everything we have covered so far has really been intended to provide you with a very firm grasp of network protocols and header dissection. While we have started to make some steps toward finding evil through BPF filtering, in this book we will work to master signature-based tools. To do so, we will make use of the most widely deployed IDS in the world: Snort (or the commercial equivalent, FirePOWER).

Before we jump into the world of signature detection, however, we are going to do a bit of research into a packet-crafting tool, Scapy, and round out our Wireshark abilities with some file extraction and forensic analysis work!

Scapy

- Powerful tool to craft and read packets
- Can manipulate all layers of TCP/IP stack
- Can easily craft application data
- Can read, manipulate, and write pcap files
- Command-line interactive interface available for simpler tasks
- Can import Scapy packages in Python for more complex tasks

Scapy is an extremely powerful tool with many diverse uses. It can be employed to test IDS/IPS rules, as well as to examine firewall access or any other packet crafting tasks you have. One of the benefits of learning Scapy is that its syntax logically follows the layering of the protocols so that learning to use it reinforces general concepts of TCP/IP.

Scapy is a robust tool that can do just about anything you can envision for packet manipulation. It may have a steeper learning curve than command-line tools such as nmap and hping3, but when you understand it and figure out what it can do, it is well worth the time investment. It enables you to craft or manipulate any field in all layers of supported protocols. It also provides support for many different application layer protocols, making it easier to read, write, or alter them.

Scapy can be run from an interactive command-line interface. This is a good way to become familiar with it. You can find many online tutorials about Scapy from its author, Philippe Biondi. He's given many conference presentations that are available online and will take you through the many wonders of Scapy. If the tasks you need to perform are more complex and require some programming logic, it may be easier to write a Python program that imports the Scapy packages and their powerful functions.

Analysts Need Packet Crafting Tools

Sends a TCP segment from inside the network to C2 server:

- The IP header IP identification value is 666.
- The TCP header has the following:
 - The SYN flag is set.
 - The source port is always 12345.
 - The destination port is 666.
 - The sequence number is 0.
 - The acknowledgement number is 88888888.
- There is payload:
 - "Sending data to" beginning at offset 0

Why would you care about the ability to craft packets as an analyst? Imagine that you learn about a new version of a *Command and Control (C2) channel* named SYNister that you'd like to detect using a Snort rule. Coincidentally for us, the characteristics of this contrived C2 are straightforward, distinct, and easily expressed in a Snort rule. The distinctive features of the traffic reflect the C2 author's relative lack of sophistication about writing malware that blends in with normal traffic. That is good for us as analysts.

A controlled host signals to the C2 server that it is preparing to send or exfiltrate data with a single TCP SYN packet that has a value of 666 in the IP identification field of the IP header. The TCP header has the SYN flag set, a source port of 12345, a destination port of 666, an abnormal TCP sequence number of 0, and an abnormal TCP acknowledgement value of 88888888. Finally, there is data associated with the SYN packet. The controlled host signals that it will send data to a specific IP address with a message beginning with "Sending data to," followed by the IP address. We won't worry about the IP address part because it can be variable.

There is some associated software on the commander's side that can accept and interpret this SYN and prepare for the receipt of data.

You Code Up a Snort Rule

- You need to generate a packet that can test/trigger the rule.
- We examine the process of doing this in Scapy.

```
alert tcp $HOME_NET 12345 -> $EXTERNAL_NET 666
(msg:"SYNister command and control";
 seq: 0; ack: 88888888; id: 666; flags: S;
 content: "Sending data to", offset 0, depth 15;
 sid:6666666;)
```

Don't let this freak you out! We will write Snort rules later.

You code up a simple Snort rule shown in the slide. Don't worry if you don't understand the syntax because we will cover Snort rules later in this book.

It examines TCP traffic from any IP address on the protected network destined for the unprotected network, from source port 12345 to destination port 666. It looks for a TCP sequence number of 0, a TCP acknowledgement number of 88888888, an IP ID value of 666, a SYN flag set, and the content of "Sending data to" beginning at offset 0 in the payload. Because this is a single packet, we do not need to deal with an established session.

As may happen many times after you write a rule, you'd like to test that your rule works. However, there is no sample pcap available, so you either need to take a chance and add it to the existing rules untested or craft up some traffic to test it. We'll use the method of crafting our traffic and writing it to a pcap. This makes it easier to run the rule multiple times without having to craft and send the packet anew each time.

You should be cognizant that it may not be optimal to have the same person write the rule and craft the traffic to trigger the rule, if at all possible. Your interpretation of the characteristics of the C2 and translation into an accurate rule may not be correct. It is possible that you may replicate that same flawed logic in crafting traffic. This can be avoided if there is another analyst available to craft the traffic or write the rule. You can better interpret the traffic and write an accurate rule between the two of you. It's kind of like editing your own writing sometimes; you read what you think you wrote and not what you actually wrote.

We're going to examine how to craft a packet using Scapy that will test the new rule.

Discovering Fields in a Given Layer: ls ()

```
>>> ls(Ether)
dst      : DestMACField      = (None)
src      : SourceMACField    = (None)
type     : XShortEnumField   = (0)
>>>
```

```
>>> ls(IP)
version  : BitField          = (4)
ihl      : BitField          = (None)
tos      : XByteField         = (0)
len      : ShortField        = (None)
id       : ShortField        = (1)
flags    : FlagsField        = (0)
frag     : BitField          = (0)
ttl      : ByteField         = (64)
proto    : ByteEnumField     = (0)
chksum   : XShortField       = (None)
src      : Emph              = (None)
dst      : Emph              = ('127.0.0.1')
options  : PacketListField   = ([])
>>>
```

```
>>> ls(TCP)
sport    : ShortEnumField    = (20)
dport    : ShortEnumField    = (80)
seq       : IntField         = (0)
ack       : IntField         = (0)
dataofs   : BitField         = (None)
reserved  : BitField         = (0)
flags     : FlagsField       = (2)
window    : ShortField       = (8192)
chksum    : XShortField      = (None)
urgptr    : ShortField       = (0)
options   : TCPOptionsField  = ({}))
>>>
```

Creating packets with Scapy isn't difficult, but we should understand some principles about its function. Scapy is an object-oriented packet crafting framework with very strong hooks in Python. This means that we can treat packets and packet headers as objects. What does this mean?

The **ls(objectname)** command from the Scapy interface can help to answer this question. It dumps the supplied object's field names, field types, and default values. We've chosen to dump the **Ether()**, **IP()**, and **TCP()** objects above, yet there are many more Scapy objects available for use. The Scapy interactive command **ls()** displays all the objects supported by Scapy. The command **lsc()** shows all the commands available.

You must first check if a given protocol is supported before trying to craft traffic. If it is, you need to list the object's attribute names so that you can refer to them.

Assigning Values

```
>>> ip=IP()
>>> ip.src="192.168.11.11"
>>> ip.dst="1.1.1.1"
>>> ip.id=666
>>> ip
<IP id=666 src=192.168.11.11 dst=1.1.1.1 |>
```

```
>>> ip2=IP(src="192.168.11.11", dst="1.1.1.1", id=666)
>>> ls(ip2)
version      : BitField          = 4          (4)
ihl          : BitField          = None         (None)
tos          : XByteField        = 0            (0)
len          : ShortField        = None         (None)
id           : ShortField        = 666          (1)
flags        : FlagsField        = 0            (0)
frag         : BitField          = 0            (0)
ttl          : ByteField         = 64           (64)
proto        : ByteEnumField     = 0            (0)
chksum       : XShortField       = None         (None)
src          : Emph              = '192.168.11.11' (None)
dst          : Emph              = '1.1.1.1'   ('127.0.0.1')
options      : PacketListField  = []          ([])
>>> █
```

Now that you have learned the protocol object and attribute names, you can start to build the packet. As discussed, Scapy uses a layered approach much like encapsulation. First, you must *instantiate* an object by assigning your own name to it. For instance, we instantiate the **IP()** object with our name of **ip**. Scapy uses Python, which is case-sensitive, permitting it to distinguish between **IP** and **ip**.

We've now got our own IP object, **ip**, where we can assign values to its attributes. The period between the object and attribute name delimits the two, as in **ip.src**. A source and destination IP address and the IP ID value are assigned using this method. The IP address is a string and must be enclosed in quotes. After we build our **ip** object, we can display some of the values in the IP header simply by referencing the object **ip**. The attribute field names and values listed are those that you have changed. Scapy has default values for each of its objects that remain unless changed.

A second way of building our IP header is to instantiate the object and assign all the attribute values at the same time, as done with **ip2**. We list it with **ls(ip2)**, which gives you more detailed information about the header, including the changed and default values. Scapy can determine the source and destination MAC addresses, as well as the source IP address when you supply the destination IP address as you craft frames to be sent over the network. It has access to the host's routing table, permitting it to assign those values so you do not have to.

About Assigning Values

Scapy errors usually occur when you save or send packets!

- Consider these examples:
 - `icmp = ICMP(type=8, code=0, seq="412", id=1)`
 - `icmp2 = ICMP(type=8, code=0, seq=412, id=1)`
- While these look the same, they aren't.
- Scapy will throw a terrifying error when you try to save or send this—why?
- Notice the first assignment assigns a string to `seq`, not an integer!

It is good to remember that Scapy is really running inside of a Python REPL (Read, Execute, Print Loop). We can forget this at times because of the ability to do things like stack layers, which we will look at in a moment. However, since it is a Python REPL, things must be of the proper types or Python will get grumpy.

A common error that is easy to make is to make an assignment of a numeric value but to put the number in quotation marks. Neither Scapy nor Python will complain about this at the time you do it, but when you attempt to send that packet over the network, write that packet to a file, or open that packet in Wireshark, Scapy will explode. When you have a mysterious error like this, a great way to troubleshoot it is to use the `ls()` function to look at the packet you are trying to send, save, or view. Look carefully at the values that have been assigned, looking specifically for quotation marks around numbers.

A similar mistake, but one that will generate an error immediately, is to attempt to assign an IP address to a field without using quotation marks. Hosts and addresses must have quotation marks around them.

Stacking Layers

```
>>> ip=IP(src="192.168.11.11", dst="1.1.1.1", id=666)
>>> tcp=TCP(sport=12345, dport=666, seq=0, ack=888888888, flags="S")
>>> pay="Sending data to 11.11.11.11"
>>> SYN=Ether()/ip/tcp/pay
>>> SYN
<Ether type=0x800 |<IP id=666 frag=0 proto=tcp src=192.168.11.11 dst=1.1.1.1 |
<TCP sport=12345 dport=666 seq=0 ack=888888888 flags=S |<Raw load='Sending data
to 11.11.11.11' |>>>>
>>> █
```

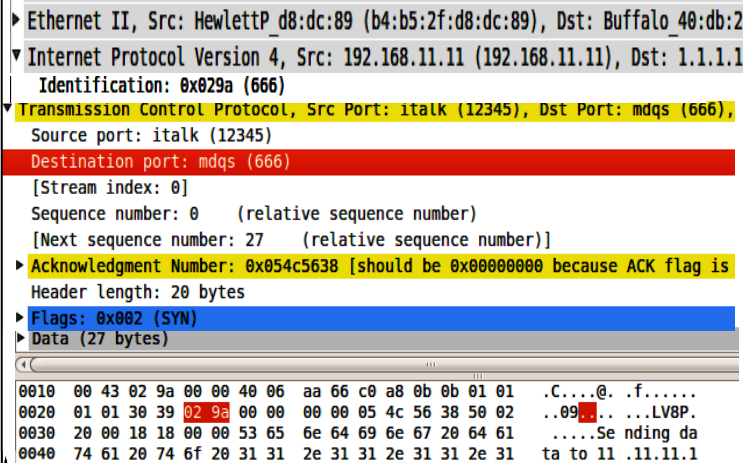
The next step in creating our SYNister packet is to instantiate all the protocol layers used and assign the attribute values that we would like. We instantiate a TCP object named **tcp** next and define the attribute values that are associated with the SYNister C2. Lastly, we assign a payload **pay** value that might be seen with the SYNister C2.

With the layers completed, we next build the packet by stacking the layers. We assign our frame a name of **SYN** and build it using the default **Ether()** object, followed by our **ip**, **tcp**, and **pay**. The slash is used to divide each layer. We didn't need to define our own **Ether()** object because Scapy learns the source and destination MAC addresses from the host's configuration files and routing tables and later assigns the values when the frame is sent. It assigns the **Ether.type** based on the protocol you supply after it, where 0x800 is IPv4.

And that brings up an interesting point: Scapy will create the packet according to the objects and order that you craft. For instance, if you reversed the IP and TCP layers as you built the frame, Scapy would create it exactly in that manner. Scapy supplies the objects and attributes but is not aware of how they are to be used. Chances are that this malformed packet would not get very far.

Writing the Record to a pcap

```
>>> wrpcap("/tmp/data-on-syn.pcap", SYN)
>>> wireshark(SYN)
```



```

▶ Ethernet II, Src: HewlettP_d8:dc:89 (b4:b5:2f:d8:dc:89), Dst: Buffalo_40:db:2
▼ Internet Protocol Version 4, Src: 192.168.11.11 (192.168.11.11), Dst: 1.1.1.1
  Identification: 0x029a (666)
▼ Transmission Control Protocol, Src Port: italk (12345), Dst Port: mdqs (666),
  Source port: italk (12345)
  Destination port: mdqs (666)
  [Stream index: 0]
  Sequence number: 0 (relative sequence number)
  [Next sequence number: 27 (relative sequence number)]
  Acknowledgment Number: 0x054c5638 [should be 0x00000000 because ACK flag is
  Header length: 20 bytes
  Flags: 0x002 (SYN)
  Data (27 bytes)
  0010 00 43 02 9a 00 00 40 06 aa 66 c0 a8 0b 0b 01 01 .C....@. .f.....
  0020 01 01 30 39 02 9a 00 00 00 00 05 4c 56 38 50 02 ..09... ..LV8P.
  0030 20 00 18 18 00 00 53 65 6e 64 69 6e 67 20 64 61 .....Se nding da
  0040 74 61 20 74 6f 20 31 31 2e 31 31 2e 31 31 2e 31 ta to 11 .11.11.1

```

SANS



data-on-syn.pcap

SEC503: Intrusion Detection In-Depth 12

We can test the Snort rule by running it against live traffic or in readback mode from a pcap. The latter is better if we anticipate having to test this multiple times.

The easiest way to save this packet is to write it to a pcap file. The Scapy **wrpcap()** command accomplishes this by supplying the pcap filename first, **/tmp/data-on-syn.pcap**, and the object or list that you want to write, **SYN**, as was demonstrated in the previous slide. A helpful Scapy capability is to invoke Wireshark so that you can display your packet(s). This is a good way to debug the packets before writing/sending them.

Note that the command shown on the slide is a continuation from the previous slide. The Scapy code from the previous slide creates the packet known as **SYN**.

Run Your Snort Rule Using Your Generated pcap

We can test out our new rule with our generated packet(s):

- `snort -q -c /etc/snort/snort.lua -r /tmp/data-on-syn.pcap`

If all goes well, we can verify that an alert is generated:

- `[**] [1:6666666:0] SYNISTER command and control [**] [Priority: 0] {TCP} 192.168.11.11:12345 -> 1.1.1.1:666`

This gives us some assurance that our rule might work when we are attacked.

Finally! We can run our new rule by Snort (or whichever network monitoring tool we are using) using our new crafted **/tmp/data-on-syn.pcap**. If all has gone well, we will see an alert appear in our logs, indicating that our packet was created correctly and that our rule appears to be functioning correctly. This gives us some measure of assurance that, should someone attack us with this new exploit, or C2 channel in this case, we have some measure of detection capability.

It is possible that you are thinking to yourself, "Okay, that's fine, but the rules that come from the vendor supplied rule feeds will detect these zero-day threats, so why do I need to bother?" This is a fair question. We will elaborate on this more later on in this book. For now, consider these two issues:

- The rules in the rule feed are okay, but by no means complete; they often are able to detect only the simplest form of any particular attack.
- We may have custom tools, software, and other things on our network that have vulnerabilities that are not generally known and certainly not going to show up in a public rule feed.

Section 1: Fundamentals of Packet Analysis	Section 2: Signature-Based Detection	Section 3: Packet Crafting	Section 4: Packet Analysis	Section 5: Packet Analysis
- An immersion class in speaking packets, protocols, and hex - Ethernet - IPv4 & IPv6 - TCP, UDP, & ICMP - Wireshark & tcpdump fundamentals, filtering, & investigation	- Scapy for building tools and simulating signatures & behaviors - Snort & Suricata - Researching common application protocols	- Architecting for massive packet capture & analysis - Overcoming the limitations of signature-based tools - Basic Zeek usage and logs - Threat modelling - Advanced correlation using Zeek	- Guided scenarios to reinforce all skills - Elaboration on effective deployment, collection, and analysis at large scale - Moving beyond alerts to data, driven analysis, & machine learning	Challenge questions guide/follow expert analysts' progression through real-world data

- Packet Crafting for IDS/IPS
- Wireshark Part III
- Application Protocols with Snort and Suricata

Signature-Based Detection

Fundamentals of Scapy and Assembling a Packet

Writing, Reading, and Altering Packet Captures

Let's leave that example Snort rule behind and examine some other ways to leverage Scapy—specifically, reading, writing, and altering pcap files. It's not listed on the slide, but we will also examine sending crafted packets!

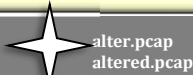
Reading and Altering a pcap (I)

```
>>> packet=rdpcap("alter.pcap")
>>> packet[0]
<Ether dst=00:50:56:f4:4f:b1 src=00:0c:29:00:28:18 type=IPv4 |<IP version=4 ihl=5 tos=0x0 len=80
id=1 flags= frag=0 ttl=64 proto=tcp checksum=0xd863 src=213.42.11.4 dst=172.16.22.5 |<TCP
sport=12322 dport=smtp seq=0 ack=0 dataofs=5 reserved=0 flags=A window=8192 chksum=0xbdcf urgptr=0
|<Raw load='Email with really sensitive information!' |>>>>
>>> packet[0][IP].src="1.1.1.1"
>>> packet[0].load="Email with really XXXXXXXXXX information!"
>>> packet[0]
<Ether dst=00:50:56:f4:4f:b1 src=00:0c:29:00:28:18 type=IPv4 |<IP version=4 ihl=5 tos=0x0 len=80
id=1 flags= frag=0 ttl=64 proto=tcp checksum=0xd863 src=1.1.1.1 dst=172.16.22.5 |<TCP sport=12322
dport=smtp seq=0 ack=0 dataofs=5 reserved=0 flags=A window=8192 chksum=0xbdcf urgptr=0 |<Raw
load='Email with really XXXXXXXXXX information!' |>>>>
>>> wrpcap("altered.pcap", packet)
```

We changed data...
Bad checksums!

```
root@sec503:/sec503/Exercises/Day3# tcpdump -r altered.pcap -n -v -t
reading from file altered.pcap, link-type EN10MB (Ethernet)
IP (tos 0x0, ttl 64, id 1, offset 0, flags [none], proto TCP (6), length 80, bad cksum d863 (->b690)!)
1.1.1.1.12322 > 172.16.22.5.25: Flags [.], cksum 0xbdcf (incorrect -> 0xfc5e), seq 0:40, ack 0, win
8192, length 40: SMTP, length: 40
Email with really XXXXXXXXXX information![!smtp]
```

SANS



SEC503: Intrusion Detection In-Depth 15

There are other great reasons for wishing to craft or modify packets. Imagine that you have some curious packets that you aren't able to make sense of. Alternatively, imagine that you have found some very interesting packets related to a new attack. In either case, you decide that you'd like to share them with the community to get assistance either in analyzing the packets or in assisting the community to identify a new threat. Imagine that the packet in the slide is one of these packets.

When we load it into Scapy, we have the ability not only to look at the packet but also to edit it! In this case, we decide that the source IP address, perhaps, reveals something sensitive. We also notice that there's a sensitive piece of information sitting in the payload. We can simply access the different fields within Scapy and edit them directly!

Notice that we tell Scapy (really, Python at this point) to access the first packet (at offset zero) in the list of packets that was loaded and to access the IP header within it. More specifically, we use a dot (.) to tell Python to access the "src" attribute within the header, allowing us to assign a new source IP address.

We also want to change the payload. Once again, we access the packet at offset zero, specifying the "load" portion of the packet (note that you do not want to try to access "Raw," even though it appears there!), and we replace the text with new text. Finally, we write the new pcap out and then check it using tcpdump. Unfortunately, we immediately notice a problem! The IP and TCP checksums are no longer valid!

Reading and Altering a pcap (2)

```
>>> packet=rdpcap("altered.pcap")
>>> del packet[0][IP].chksum
>>> del packet[0][TCP].chksum
>>> wrpcap("altered-fixed.pcap", packet)
```

Don't set the checksum to zero!
You must delete it to force Scapy to recalculate.

```
root@sec503:/sec503/Exercises/Day3# tcpdump -r altered-fixed.pcap -n -v -t
reading from file altered-fixed.pcap, link-type EN10MB (Ethernet)
IP (tos 0x0, ttl 64, id 1, offset 0, flags [none], proto TCP (6), length 80)
  1.1.1.1.12322 > 172.16.22.5.25: Flags [.], cksum 0xfc5e (correct), seq
  0:40, ack 0, win 8192, length 40: SMTP, length: 40
    Email with really XXXXXXXXXX information![!smtp]
```



How can we fix the bad checksum? It's actually quite simple, but it's important that you understand the proper way to do it. Most people first try to set the checksum values to zero. It's understandable that this mistake is made because when you look at a raw IP, TCP, UDP, ICMP, or another header that supports a checksum, Scapy will tell you that the default value is zero! Unfortunately, if you manually *set* the checksum to zero, Scapy now assumes that you *want* it to be the literal value "0"!

The correct way to tell Scapy to recalculate the checksum when the packet is sent or stored is to delete the checksum. The keyword "del" can be used for this purpose, as pictured in the slide.

Sending Packets

Must be root to send a packet

```
>>> packet=IP(dst="192.168.11.1")/TCP(sport=1024, dport=80, flags="S")
>>> send(packet)
.
Sent 1 packets.
>>> sr1(packet)
Begin emission:
Finished to send 1 packets.
*
Received 1 packets, got 1 answers, remaining 0 packets
<IP version=4L ihl=5L tos=0x0 len=44 id=0 flags=DF frag=0L ttl=64 proto=tcp chksu
m=0xa363 src=192.168.11.1 dst=192.168.11.23 options=[] |<TCP sport=http dport=102
4 seq=757387650 ack=1 dataofs=6L reserved=0L flags=SA window=5840 checksum=0xe6e5 ur
gptr=0 options=[('MSS', 1460)] |<Padding load='\x00\x00' |>>>
>>>
```

If you want to test your Snort rule against live traffic, you could send the packet instead of writing it to a file. The crafted packet on the slide is unrelated to the SYNister C2. This demonstration requires an actual reachable destination IP address to send traffic. Our new packet is sent using the **send** command, which sends a single packet. Scapy creates the Ethernet layer when sending to the network, so we do not have to.

Let's look at the **sr1()** command. This sends the packet and listens for a response by detecting an expected match of the IP address, protocol, ports, and flags. This is a feature of Scapy that distinguishes it from command-line crafting tools such as hping3. They can send packets, and perhaps even display a response, yet they cannot present the received packet to you as done here.

Why is this important? If you need to craft a TCP session involving the three-way handshake, you need to acknowledge the server's sequence number in the third exchange of the three-way handshake. This is done by crafting and returning a value that is one more than the server's sequence number.

TCP "Interference"

```
>>> ip=IP(src="192.168.1.104", dst="192.168.1.103" )
>>> tcp=TCP(sport=1024,dport=80,flags="S",seq=10)
>>> srl(ip/tcp)
Finished to send 1 packets.
Received 2 packets, got 1 answers, remaining 0 packets
<IP  version=4L ihl=5L tos=0x0 len=44 id=0 flags=DF frag=0L ttl=64
proto=tcp chksum=0xd293 src=192.168.1.103 dst=192.168.1.104 options=''
|<TCP  sport=www dport=1024 seq=3651161442L ack=1 dataofs=6L reserved=0L
flags=SA window=5840 chksum=0xedb9 urgptr=0 options=[('MSS', 1460)]
|<Padding  load='\x00\x00' |>>>
```

```
tcpdump -s0 -nt 'host 192.168.1.103 and tcp'
192.168.1.104.1024 > 192.168.1.103.80: Flags [S], seq 10, length 0
192.168.1.103.80 > 192.168.1.104.1024: Flags [S.], seq 3651161442,
11, length 0
192.168.1.104.1024 > 192.168.1.103.80: Flags [R.], seq 1, length 0
```



Where does this reset come from?

Let's examine some universal unintended consequences of attempting to establish a TCP session. We craft an IP header where the sending IP address is 192.168.1.104 and the receiving host is 192.168.1.103. The TCP header is crafted with the SYN flag set. We then use the Scapy function `srl()` to send and listen for the corresponding SYN/ACK. A matching packet is returned from the address 192.168.1.103 with the SYN and ACK flags sent as expected. So far, so good.

But listening with `tcpdump` in another terminal, we see another packet. The sending host 192.168.1.104 resets the connection. This just closed the session we were attempting to establish. This just prevented us from emulating the client side of the session. What's causing this?

A UDP connection to the destination host would have a similar result upon receiving a UDP response. Instead of a TCP reset, the sending host would generate an ICMP **port unreachable**.

(Image credit: <https://www.redbubble.com/people/dmitriylo/works/23730940-game-over-pixel-art-invader?p=poster>, retrieved November 2018)

Cooked versus Raw Sockets

- Cooked socket uses system's native TCP/IP stack
 - Kernel builds the packet
 - Assigns appropriate IP/TCP/UDP... header values
 - You supply the payload.
- Raw Socket circumvents TCP/IP stack
 - You build the packet.
 - You assign header values.
 - You supply the payload.

The answer to the "interference" lies in the way we are sending the traffic. Typically, a network socket is used to define, set up, and take care of sending and receiving network traffic. This is something that the operating system normally takes care of. For instance, when you use your browser to navigate the internet, the browser software interacts with the operating system to define and use sockets for TCP data transmission. This type of socket is known as a *cooked socket* because it interacts directly with the operating system kernel to assign the appropriate IP, UDP, and TCP header values. The payload is added by the browser depending on what you enter, click on, etc. A cooked socket uses the native TCP/IP stack, and sent packets are paired with received responses. Scapy does not use cooked sockets because cooked sockets don't allow you to alter any of the protocol header values.

On Linux, Scapy uses the *PF_PACKET* protocol family when interacting with sockets. *PF_PACKET* permits Scapy to send and retrieve packets directly to the network card driver. This circumvents the TCP/IP stack and is sometimes also called a *raw* socket. This is what allows us to build the packets as we've done. It allows us control over the header values for IP, TCP, UDP, etc. And, we supply the payload we want too. The "interference" we saw in the previous two slides was the TCP/IP stack complaining that it didn't send the packets we crafted. And it's correct; it didn't send them so it should complain when it receives the responses from the other host.

Does this mean that we can't emulate a TCP client or server or attempt to carry out a multi-packet UDP conversation? We need to find a way to address the issue of the native TCP/IP stack interfering with our crafting efforts.

iptables to the Rescue

TCP - Block outbound resets:

```
root@desktop: iptables -A OUTPUT -p tcp --tcp-flags RST RST -j DROP
```

UDP - Block outbound ICMP port unreachables:

```
root@destktop: iptables -A OUTPUT -p ICMP --icmp-type port-unreachable -j DROP
```

You could be more precise if you wished.

Here's the syntax to use to block outbound TCP resets and ICMP unreachables for UDP traffic when invoking from the command line.

The first iptables statement identifies that the direction for blocking is OUTPUT and the protocol is TCP, where the TCP flag to examine is RESET and the TCP flag that must be set is RESET. If these conditions match, the packet will be forwarded to the DROP rule.

To prevent an ICMP port unreachable message for UDP traffic, we use iptables again. This time, we identify a different protocol, ICMP, and specify the ICMP type of port-unreachable that we want iptables to DROP.

You could be more precise when writing these rules. For instance, if you only wanted to drop RST packets related to a specific port, you could certainly add that. During development of a new Scapy script, however, we generally find it much easier to simply drop all RST packets.

Scripting with Scapy

You can automate any network behavior

- Sample scripts on VM
 - LaBrea implementation
 - Four-way handshake
 - Urgent pointer research
 - IP fragment experiments
 - Anonymizing packets

```
import sys
import random
from scapy.all import *

version = 0.2
def determineMACAddress():
    localMACs = [get_if_hwaddr(i) for i in get_if_list()]
    # Assume the last one is the MAC to send from.
    return localMACs[-1]

def spoofIsAt(pkt):
    isAt = ARP()
    isAt.hwdst=pkt[ARP].hwsrc
    isAt.pdst=pkt[ARP].psrc
    isAt.psrc=pkt[ARP].pdst
    isAt.hwsrc=sourceMAC
    isAt.op=2
    #is-at print("Taking over {0}!".format(isAt.psrc))
    send(isAt, verbose = 0)
...
```

The command-line interface is convenient if you want to send simple packets or you're not doing anything that involves complex logic. However, if you're using it to generate simple packets, you could use any other packet crafting tool and not have to learn Scapy. Why is Scapy such a big deal?

The real power of Scapy is the ability to write scripts that can both send packets and receive packets, reacting to the incoming packets. For example, in the TCP section in Book 2, we mentioned the possibility of a four-way handshake. Does this really work? What does it look like? We can experiment with this behavior by creating a simple script to act as either the client or the server in a four-way handshake.

Similarly, we discussed the window size field in the TCP section as well. This is used for flow control in a TCP connection. An old (but still very useful) technique was implemented in a tool called "LaBrea Tarpit." This tool manipulated the window size of spoofed packets, causing malware and other scanning tools to "get stuck" when the window size is set to zero. While that tool has been unmaintained for a very long time, reimplementing it in Scapy is easy. In fact, we have a modern version of that tool in the `/sec503/ShowMeThePackets/Scapy` directory on the VM. Most likely, your instructor will discuss some of the details of this tool and demonstrate its use. We'll give you some details on how this window size manipulation works on the next two pages.

There are a variety of other useful scripts available in that directory as well, some of which will come up later in our course. While we are not expecting you to write Python scripts in this class, you should be aware of Scapy. If you want to go further with it, it would be worth your time to invest some effort in learning to code basic Python scripts.

- Do you have some unused network space? Let LaBrea take over unused segments.
- *Any* connections to LaBrea are anomalous!

When **LaBrea** is installed on a host on a network, it first listens for ARP requests to unassigned IP numbers. It fakes a response to this. If a SYN follows from the scanning host (in this case, usually an infected host), the **LaBrea** host fakes a SYN/ACK response. **LaBrea** can be used against any TCP scan or attempted TCP connection to an unassigned IP number. After receiving the faked SYN/ACK response, the scanning host then completes the three-way handshake and attempts to send some data. The **LaBrea** host never ACKs the data sent by the scanning host. Thus, the scanning host is "tarpitted" in retransmissions until it times out. This consumes resources on the scanning host and slows its capability to scan, especially if it waits for a response to proceed with further scanning.

```
00:00:00.000000 ARP, Request who-has 10.50.7.220 tell 10.50.7.1, length 46
00:00:01.002625 ARP, Request who-has 10.50.7.220 tell 10.50.7.1, length 46
00:00:01.008106 ARP, Reply 10.50.7.220 is-at 00:0c:29:07:23:bd, length 28
00:00:01.009495 IP 10.50.7.1.60094 > 10.50.7.220.80: Flags [SEW], seq 4049581916, win 65535
00:00:01.009570 IP 10.50.7.1.60094 > 10.50.7.220.80: Flags [S], seq 4049581916, win 65535
00:00:01.012780 IP 10.50.7.220.80 > 10.50.7.1.60094: Flags [S.], seq 1182748721, ack
4049581917, win 54, length 0
00:00:01.013188 IP 10.50.7.1.60094 > 10.50.7.220.80: Flags [.], ack 1, win 65535, length 0
00:00:01.013195 IP 10.50.7.1.60094 > 10.50.7.220.80: Flags [.], seq 1:55, ack 1, win 65535,
length 54: HTTP: GET /etc/passwd HTTP/1.1
00:00:01.023452 IP 10.50.7.220.80 > 10.50.7.1.60094: Flags [.] ack 1, win 0, length 0
00:00:01.026596 IP 10.50.7.220.80 > 10.50.7.1.60094: Flags [.] ack 55, win 0, length 0
00:00:06.030656 IP 10.50.7.1.60094 > 10.50.7.220.80: Flags [.], seq 55:56, ack 1, win 65535,
length 1: HTTP
00:00:06.036180 IP 10.50.7.220.80 > 10.50.7.1.60094: Flags [.] ack 55, win 0, length 0
00:00:11.034219 IP 10.50.7.1.60094 > 10.50.7.220.80: Flags [.], seq 55:56, ack 1, win 65535,
length 1: HTTP
00:00:11.043366 IP 10.50.7.220.80 > 10.50.7.1.60094: Flags [.] ack 55, win 0, length 0
00:00:16.038796 IP 10.50.7.1.60094 > 10.50.7.220.80: Flags [.], seq 55:56, ack 1, win 65535,
length 1: HTTP
```

Notice in the slide that the system attempting to talk to LaBrea sends 1-byte window probes immediately but then begins to back off.

Packet Crafting for IDS/IPS Wrap-Up

Several ways to craft traffic:

- Command-line tools that are restrictive
- Scapy with far more functionality

Scapy is unique because:

- It enables you to change any value in any header or payload.
- It can listen for a matching response to a sent packet.
- It is Python based, permitting complex processing.
- It performs some "under the hood" processing, such as computing checksums on crafted packets and assigning appropriate IP and MAC addresses.

We've taken a brief and superficial look at the power of Scapy. It is so versatile and useful. The recommendation is to pursue learning more about it if you have the interest. It is superior to command-line tools because they are restrictive in what they can do.

Scapy is unique in many ways compared to other packet crafting tools and languages. You have access to any field in any header and you can supply the payload. It is one of the few, if only, tools that can listen for a matching response to a sent frame/packet and present the response to you to examine. The support of Python enables you to do complex processing. And, finally, it does some "under the hood" processing that other languages require you to do yourself, such as checksum computation and assigning appropriate Ethernet MAC addresses.

Packet Crafting for IDS/IPS Exercises

Workbook exercises "Packet Crafting for IDS/IPS"

This page intentionally left blank.

• An immersion class in speaking packets, protocols, and hex • Ethernet • IPv4 & IPv6 • TCP, UDP, & ICMP • Wireshark & tcpdump fundamentals, filtering, & investigation	Section 2: Signature-Based Detection • Scapy for building tools and simulating signatures & behaviors • Snort & Suricata • Researching common application protocols	• Architecting for massive packet capture & analysis • Overcoming the limitations of signature-based tools • Basic Zeek usage and logs • Threat modelling • Advanced correlation using Zeek	• Guided scenarios to reinforce all skills • Elaboration on effective deployment, collection, and analysis at large scale • Moving beyond alerts to data, driven analysis, & machine learning	• Challenge questions guide/follow expert analysts' progression through real-world data
---	--	---	---	---

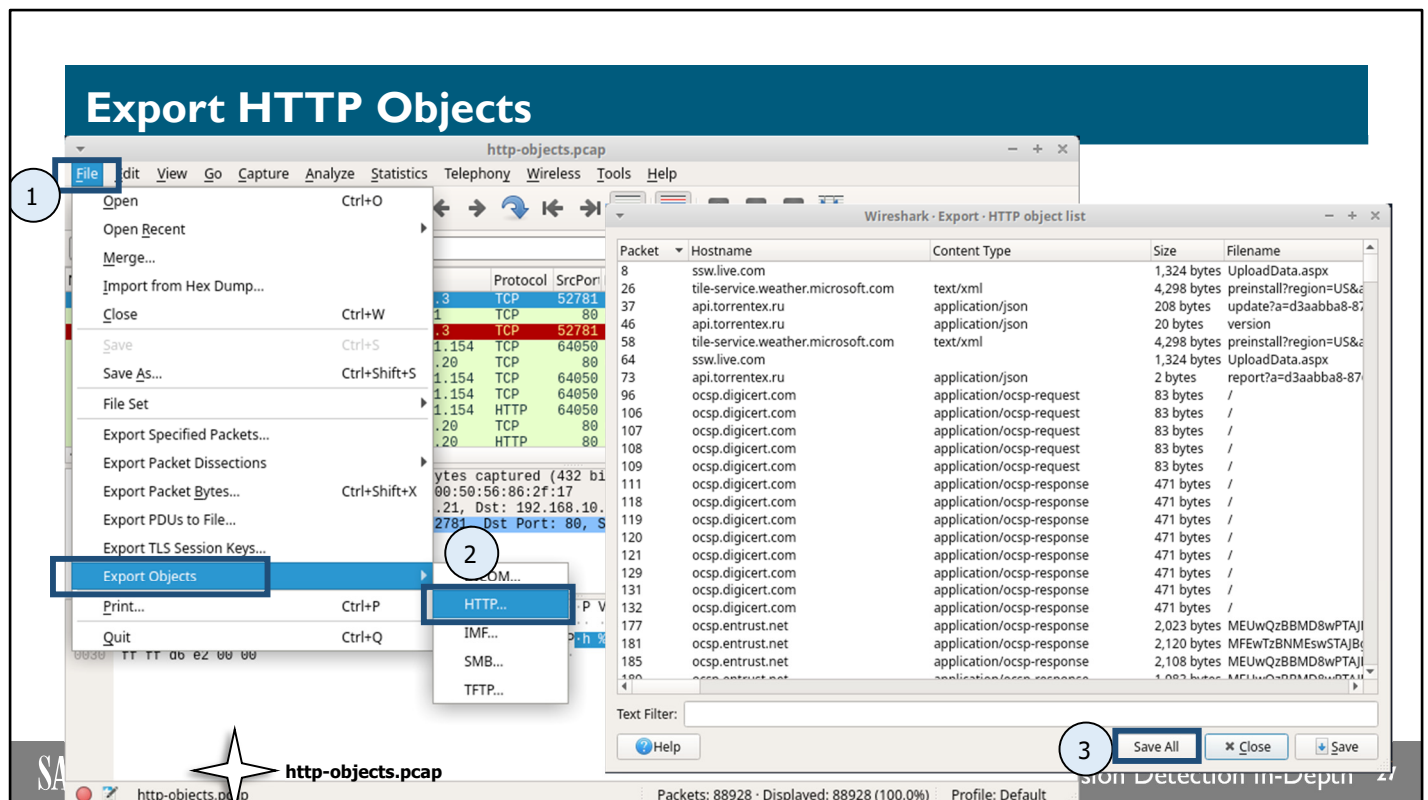
- Packet Crafting for IDS/IPS
- Wireshark Part III
- Application Protocols with Snort and Suricata

Signature-Based Detection

Extracting Files

Forensic Reconstruction

This is our final section that specifically focuses on Wireshark. Wireshark is so comprehensive that it has taken three sections just to give you an overview of its capabilities. In support of our evaluation of application protocols, we will discuss some features that Wireshark has that can export web objects and extract SMTP attachments for further scrutiny. We'll review an incident using Wireshark to demonstrate one approach to an investigation. And finally, we'll wrap up with useful topics we have not covered, such as using tshark, the command-line version of Wireshark.



Wireshark has the built-in ability to export a number of different types of file objects. To export HTTP objects, for example, navigate to the menu **File** → **Export Objects** → **HTTP**. Next, a panel will appear that has all the object names; save them all by selecting **Save All** at the bottom right of the panel. Another panel appears that asks for a name of a directory that will be created to store all the objects (in this example, **Export Objects**). Then select **OK** at the bottom of the panel.

✦ To view the output:

wireshark http-objects.pcap

File → Export Objects → HTTP (Save ALL → Enter Directory Name → OK)

Exporting Objects with Wireshark

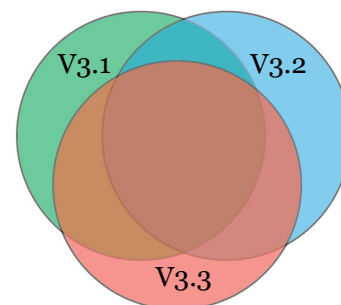
All Tools Must Lie

- The Export Objects function does a great job, as far as it goes.
- It *definitely* will miss files!
- If you are doing an investigation, this is very important to take into account.
- Don't rely exclusively on Wireshark (or any other tool!) for automatic extraction.



The feature is still useful:

- If extracting "most" files is enough
- If you know what you're looking for and it happens to be extracted



Don't forget our warning about relying too heavily on tools! In fact, Wireshark has a variety of issues revolving around extracting files.

Our experience is that if you take three different versions of Wireshark and use the **Export Objects** function with each of those versions, you will find a *different number of extracted files... every time!* Your first thought might be, "OK, but that's just because newer versions find more things, right?" The answer is No. What you'll find is that there is overlap, but the newest version will miss things that the older versions find! As a network troubleshooting tool, this doesn't matter. As a tool that might be used for an incident investigation or forensic reconstruction, this is far more concerning! This makes it obvious that Wireshark is not consistently able to identify all possible streams! Does this make you begin to question whether there are more objects than the ones that Wireshark shows?

We should also mention that some will read this and think, "Wow, okay, so we'd better use NetworkMiner (or some other similar product) instead!" This is a good thought, but *every tool has this same problem*. That is, every tool will miss some files. This means that we must have other techniques for extracting data.

There was also a very irritating bug that only became apparent if you sorted the columns in the list of possible objects to export and then selected one of those objects. Rather than exporting the object that you selected, Wireshark exported the stream that would have been on that line had you not sorted the list!

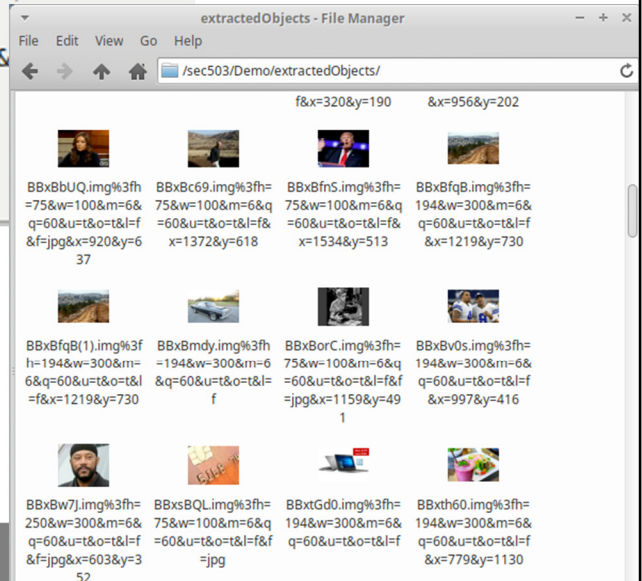
While these bugs are serious, there is another danger in relying too heavily on a tool. In this case, what happens when we have to extract a file from a protocol that Wireshark does not support with **Export Objects** or isn't supported by whichever tool you are using? We will address this shortly.

Find Executables and Other Interesting Aspects of the Exported Files

```
sans@sec503:/sec503/Demo/extractedObjects$ ls | wc -l
508
sans@sec503:/sec503/Demo/extractedObjects$ grep "^MZ" *
Binary file AmigoInstallerXPDB.exe matches
Binary file extension_0_46_0_4.cr%3fcms_redirect=yes&
AEF38C1B3CA4A5FF3BE4E2E260&key=cms1 matches
Binary file MailRuUpdater.exe matches
Binary file mpam-d.exe matches
Binary file waiter.exe matches
sans@sec503:/sec503/Demo/extractedObjects$
```



Of course, you should *never* start clicking on or opening these files unless you are on a sandbox system!



In another terminal, we look in the directory **extractedObjects** and count the number of files exported using the **wc -l** command. There are 508 files extracted. We can find the executables that were extracted using the command **grep "^MZ"** in all the files' content. **MZ** is a file header that signifies a DOS executable when these characters appear as the first 2 bytes in the file. While new file formats such as Portable Executables (PEs) have replaced the DOS format, the newer programs still use the MZ header as well.

When we look at the files in that directory using a file browser, it creates icons for us, allowing us to get a feel for what types of files are present. Realize, however, that there is some risk here! The direct and obvious risk is that you (or someone) starts double-clicking on these files and just opens them up! Unless you are on a sandbox system, that is very risky! Could these have malware of some kind? Certainly!

There is another risk, however. In case you are unaware, there have been a number of cases of flaws found within Windows Explorer and other graphical file browsers that allow an exploit to execute *simply by viewing the icon in the file browser!* Still, getting the files extracted makes them much easier to deal with since the data has all been properly reassembled by Wireshark.

Extraction without "Export Objects"

What if the protocol isn't supported or "Export Objects" missed the file you want?

- Suppose you are viewing an SMTP exchange in Wireshark.
- When following the TCP stream, an attachment is found that you would like to examine.
- MIME is used for attachments that require 8-bit formatting because SMTP supports 7-bit ASCII only.
- MIME encodes the attachment.

While HTTP, SMB, and other protocols are supported within Wireshark for easy export of transferred files, not all protocols are so lucky. What if we want to extract something that isn't supported with this feature? Alternatively, what if the automatic file extraction feature failed to extract a file that we have found through other means? Let's see.

We will look at SMTP as an example since SMTP presents some additional challenges. SMTP was developed long ago when SMTP messages and attachments were typically text-only, so SMTP supported 7-bit ASCII character representation only. But today there are many more types of messages and attachments than plain ASCII text: foreign language character sets, image files, PDF files, and PowerPoint presentations, to name a few, which are in 8-bit binary form. The *Multipurpose Internet Mail Extensions (MIME)* is an internet standard used to support formats other than ASCII.

MIME uses different headers to describe the content that it formats, and it uses a boundary separator for multipart messages. Typically, the boundary separator is a generated number that is placed before and after the MIME-encoded attachment. The boundary separator string is defined in a Content-Type header parameter. Additional Content-Type headers indicate the type of data that follows the header, such as text/plain or application/pdf. Content-Transfer-Encoding designates the format of the data, such as binary or base64. If the data is an attachment instead of embedded in the email message, it has a Content-Disposition of "attachment."

A challenge arises when you want to examine one of these attachments. More than likely, it is not in a readable format, and you have to decode the data to make sense of it. We will examine an SMTP conversation with a base64-encoded attachment, a common encoding format.

Extracting an SMTP Attachment

Wireshark enables you to save a conversation but cannot decode an attachment encoded in base64.

You need to edit the saved conversation so that all that remains is the encoded attachment.

Decode the attachment so that it is readable.

Wireshark can assist in extracting an SMTP attachment, but you must perform some additional steps to prepare the attachment for examination. You can save the **Follow** → **TCP Stream** conversation into a file. This contains the entire SMTP exchange—more than you want. Then you must edit the file that contains the saved conversation so that all that remains is the encoded content. Finally, you need to decode the content to restore it to its original format.

The example explained in the next several slides shows how to extract an attachment that is encoded in base64. There is a command known as **base64** that can be used on Linux/BSD operating systems to decode the attachment.

SMTP Conversation Stream

1 Stream Content

```
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary="-----_MIME_BOUNDARY_000_11181"
-----_MIME_BOUNDARY_000_11181
Content-Type: text/plain

This is a test mailing
-----_MIME_BOUNDARY_000_11181
Content-Type: application/octet-stream
Content-Transfer-Encoding: BASE64
Content-Disposition: attachment

JVBERi0xLjQKJ3Jfcsj6IKNSAwIG9iago8PC9MZWN5dGggNiAwIFVlRmlsdGvYIC96bGF0ZURlY29K
ZT4+CnN0cmVhbQp4nKVZ23Lb0BJ911fgLVSVhRAAR3nL0M6Wd3Z21ZVJtpx5oCna5oY1FZKy4n+Y
X/E/TjdAgABlxd\KuVy2RKDR190nW/Ep4z4+DP8zdez1x9ictfNFPIv+L2bfZsXuYAMf/I1+W0J
ixL8Ynk7U/sYiTMJU0GTLCzXmXamNANicJfzV3Ht3X5/YvHvswJ95m/8IOFL4jzYL7832zhU85g
ew7SUSB00xAPEBFZns28i7fzV6AQPGUhdVMBhy5XM+/8BLE06nCfBn5MFozjY99RgTyRd5dXr8/f
4hZ0QxSWHdR0q694370099rPs7/Wv5bncUjGkVpqldcnH4mWb0id2VeKIC5a0L7sfCktstIX5jP
33vyd1WQ26ZdZz3Jm1oKBptYqAwIaR00BjWuQI08LKU8ScFt6kHbF62yzXLG0jBBAbw2RaA3Lrv6
78XL71enV2op8wc/RL72wWj+dbPpy6bu/iLXTzk5vrz48/zy3QL+asj707MTQinFTzU5vZCfLz8u
zfcHwVCKFAYB6m/KPDVxdSpsjbr3cnV8YfT35enLxdugCEddHBpjP9AgJf3BXmnrdxk+dfsrBL
nVfbVdGRbv9WZV8W3CRFLILgDs7WPnLPCh\kpD9E34PATyXEEEFQQT3XbvmBDFSZSZX7RuaFSnuV
FhvvRSiTDuXENSOG2VRZwZuMC9Th4zoWnYahN/i60mi0ROMG65B9k+nxTGwafVTuXcRcFEiU0vY
```

MIME header for base64 attachment

2

Save As

Raw

Save

carve-smtp.pcap

SEC503: Intrusion Detection In-Depth 32

First, use the **Follow TCP Stream** to analyze the particular conversation of interest. Next, save the conversation in raw format to a filename of your choosing. The file used in the example is `/tmp/smtp-carve.txt`.

One of the pertinent parts of this SMTP message is the MIME boundary, the line that begins with dashes followed by the long number `_MIME_BOUNDARY_000_11181`. This starts a MIME message. There should be an identical string following the MIME message to separate it from any subsequent SMTP text/attachments/headers.

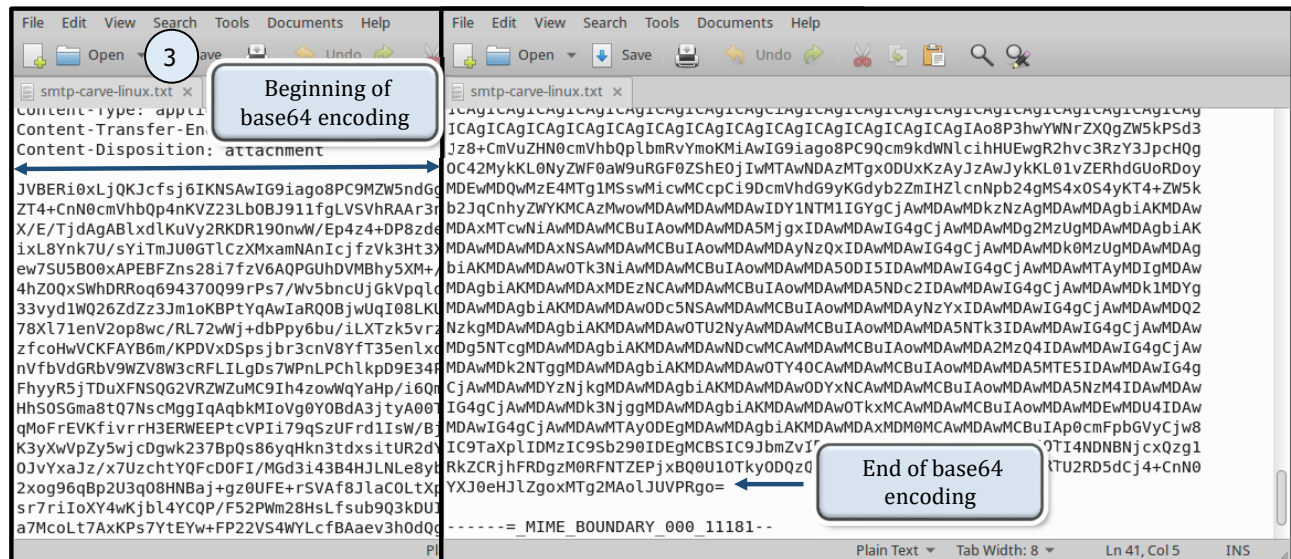
Following the MIME boundary are MIME headers. The Content-Type header indicates that it's an application/octet-stream. It is encoded in base64 and is an attachment. This is the part that we want to "carve out."

✦ To see the output, enter the following on the command line:

wireshark carve-smtp.pcap

Analyze → Follow TCP Stream → Save As (Raw) (/tmp/smtp-carve.txt)

Edit the Saved File and Remove Headers/Trailers



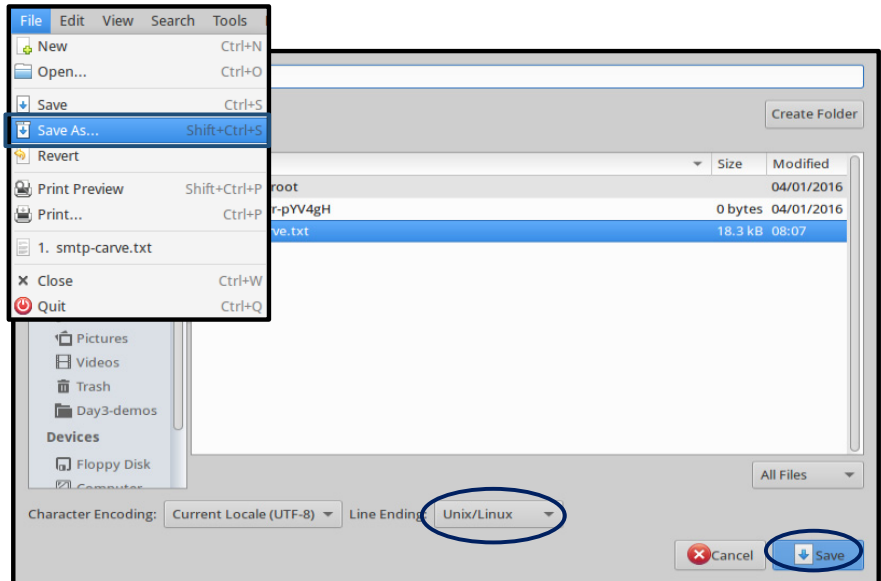
Next, you want to save the base64encoded text only. This means that you have to delete the text that is found before the start and after the end of the base64 encoding. You will know where the base64 encoding begins and ends because you see the identical boundary markers `_MIME_BOUNDARY_000_11181_`. You must delete all spaces and text following the base64 encoding.

Save the File

4

Use an editor that can change line encodings:

- (vi, vim, emacs, gedit, sublime, notepad++, etc.)
- It doesn't matter what they were...
- Save them with the encoding of the platform you are on.

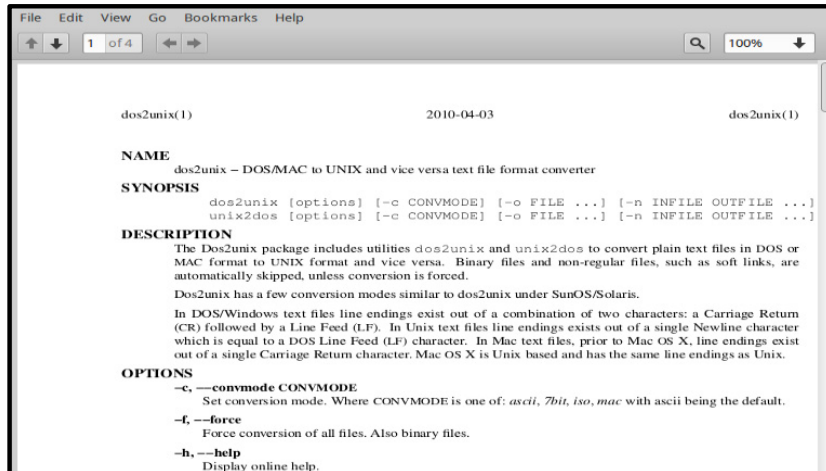


Next, either exit Wireshark or use a different terminal to edit the saved file (in this case, **/tmp/smtp-carve.txt**).

You may need to remove the **^M** characters from the file. These are a result of different line endings used in Windows and UNIX. There are a few ways to remove them, yet gedit has the capability to do this for you. In this example, assume the SMTP content has been saved to filename **/tmp/smtp-carve.txt**. Use gedit and **Save** with a **Line Ending** value of **UNIX/Linux**. The saved filename is **/tmp/smtp-carve-linux.txt**.

Use base64 to Decode

```
sans@sec503: base64 -d /tmp/smtp-carve-linux-base64.txt > /tmp/attached
sans@sec503: file /tmp/attached
/tmp/attached: PDF document, version 1.4
```



Finally, the base64-encoded text file **/tmp/smtp-carve-linux-base64.txt** is decoded using the **base64** command with the **-d** option, signifying decode. The new filename is **/tmp/attached.pdf**. It is a document containing the man page for the **dos2unix** command, which could have been used to reformat the Windows line endings with Linux line endings, as we accomplished using **gedit** and saving to a Linux format.

Section 1: Introduction to Intrusion Detection	Section 2: Signature-Based Detection	Section 3: Anomaly-Based Detection	Section 4: Host-Based Intrusion Detection	Section 5: Network-Based Intrusion Detection
• An immersion class in speaking packets, protocols, and hex • Ethernet • IPv4 & IPv6 • TCP, UDP, & ICMP • Wireshark & tcpdump fundamentals, filtering, & investigation	• Scapy for building tools and simulating signatures & behaviors • Snort & Suricata • Researching common application protocols	• Architecting for massive packet capture & analysis • Overcoming the limitations of signature-based tools • Basic Zeek usage and logs • Threat modelling • Advanced correlation using Zeek	• Guided scenarios to reinforce all skills • Elaboration on effective deployment, collection, and analysis at large scale • Moving beyond alerts to data, driven analysis, & machine learning	• Challenge questions guide/follow expert analysts' progression through real-world data

- Packet Crafting for IDS/IPS
- Wireshark Part III
- Application Protocols with Snort and Suricata

Signature-Based Detection

Extracting Files

Forensic Reconstruction

This page intentionally left blank.

General Outline of Steps to Follow

Notification of a compromised host

Use Wireshark's "Statistics" overview to see protocols and conversations.

Examine the conversations and attempt to re-create what transpired.

When you find an activity of interest, ask questions:

- What happened?
- Why did we do that?
- Lather, rinse, repeat

You get a call from the help desk that a user workstation has been compromised. Fortunately, you have full packet data captures archived that cover the approximate time of the compromise. Wireshark can help you get a high-level view of activity that occurred around that time with its **Statistics** menu options.

Wireshark also facilitates investigation of individual conversations and enables you to iteratively discover, process, and analyze conversations in an attempt to re-create what actually transpired. You can find different clues in the traffic that may cause you to pursue a particular aspect of analysis. Your discoveries may uncover something else that you may want to pursue. The way you progress in the examination is largely dependent on your logic for pursuing discoveries as well as what is revealed by Wireshark as you pursue the investigation. In other words, there is no prescribed methodology because of the pattern of different attacks.

Note: The **forensics.pcap** with the records from this investigation is used for demonstration purposes only. Timestamps on the captured records and dates and times reflected in the payload records themselves do not match. Disregard the payload timestamps, as they have nothing to do with the attack. The timestamp discrepancies result from rewriting the records. The malware that is in the pcap was altered; therefore, it cannot be extracted and analyzed.

The Scenario

CIO: We just got a call telling us that our customer data has been posted out on something called "Pastey Bins."

JSG: Yep. Sounds right.

CIO: Did it happen? I need to know right now!

JSG: Let me look into it. I'll get back to you.

To illustrate what this process might look like, let's walk through a simple scenario. Let's imagine that our organization has just received a report that customer data has been stolen or otherwise exfiltrated out of the organization. Let's also imagine that our organization has packet captures covering our border gateway for the period of time during which this exfiltration likely occurred. How would we proceed?

(Before we do proceed, we should point out to you that we are not yet halfway through the class. If you are wondering, "Okay, but how would I find the important stuff on my network?" as you work on labs, we are very definitely heading in that direction at this point.)

JSG, in this case, is the Jaded Security Girl who has been telling management that security is terrible, but no one has been listening to her. ☺

Fire Up Wireshark Statistics

Anything obviously odd? Protocols that you don't recognize or that don't belong on your network?

If not, look at the data another way.

Protocol	Percent Packets	Packets	Percent Bytes
Frame	100.0	1241	100.0
Ethernet	100.0	1241	1.2
Internet Protocol Version 6	3.4	42	0.1
Internet Control Message Protocol v6	3.4	42	0.1
Internet Protocol Version 4	96.6	1199	1.7
User Datagram Protocol	7.2	89	0.1
Simple Service Discovery Protocol	6.1	76	1.7
NetBIOS Datagram Service	0.1	1	0.0
Domain Name System	1.0	12	0.1
Transmission Control Protocol	89.4	1110	94.9
Simple Mail Transfer Protocol	7.9	98	1.1
Internet Message Format	0.6	7	0.9
Secure Sockets Layer	0.4	5	0.5
Hypertext Transfer Protocol	0.2	2	0.1

Let's get an overview of the activity of the traffic found in **forensics.pcap**, starting with the **Statistics → Protocol Hierarchy** menu selection, showing the protocols found in the traffic. The output that is shown on the slide represents typical protocols. Yet it might provide an indication of unusual or unexpected protocols on a particular network.

✦ To see the output, enter the following on the command line:

wireshark forensics.pcap

List TCP Conversations

Our Host

Large transfer out
What's in it?
Follow the stream!

Address A	Port A	Address B	Port B	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A
173.255.224.88	43100	55.9.19.52	80	840	1,266 k	840	1,266 k	0
173.255.224.59	1699	14.4.4.55	80	47	29 k	24	1,964	0
173.255.224.88	1492	14.4.4.59	80	47	29 k	24	1,964	0
65.55.111.101	57036	173.255.224.66	25	23	3,544	11	2,609	0
65.55.111.101	50182	173.255.224.66	25	23	3,544	11	2,609	0
65.55.111.101	35654	173.255.224.66	25	23	3,542	11	2,607	0
65.55.111.101	3585	173.255.224.66	25	23	3,542	11	2,607	0
65.55.111.101	40829	173.255.224.66	25	23	3,542	11	2,607	0
65.55.111.101	28444	173.255.224.66	25	23	3,542	11	2,607	0
65.55.111.101	21893	173.255.224.66	25	23	3,540	11	2,605	0
74.125.226.118	35007	173.255.224.101	55212	2	2,558	0	0	0
74.125.226.149	55212	173.255.224.101	443	2	2,557	0	0	0
74.125.226.149	443	173.255.224.101	443	2	1,482	2	1,482	0
74.125.226.181	443	173.255.224.101	443	2	238	2	238	0

There are more statistics available; we look at conversations in the **Statistics → Conversations** menu selection combination. There is an option to look at the **TCP** conversations only using a tab at the top. TCP is the most common and widely used protocol, so it's logical to start with that.

We see some common services, HTTP and SMTP, associated with ports 80 and 25. One of the first places to start is by examining the number of bytes exchanged in the sessions. You see a heading labeled **Bytes** that can be sorted to reveal the session exchanges with the greatest number of bytes, potentially exposing an aberration in volume. Click twice on the **Bytes** heading itself to cause Wireshark to first sort in ascending then descending order.

The monitored network is 173.255.224.0/24. What stands out among the connections is the first one after sorting from largest to smallest between 173.255.224.88 and 55.9.19.52 over port 80. The client sends 840 packets with 1,266,352 bytes, whereas the server returns 0 packets and 0 bytes. In reality, the server does send packets and bytes, yet with no payload, just acknowledgements of the client's data. Wireshark counts payload bytes only.

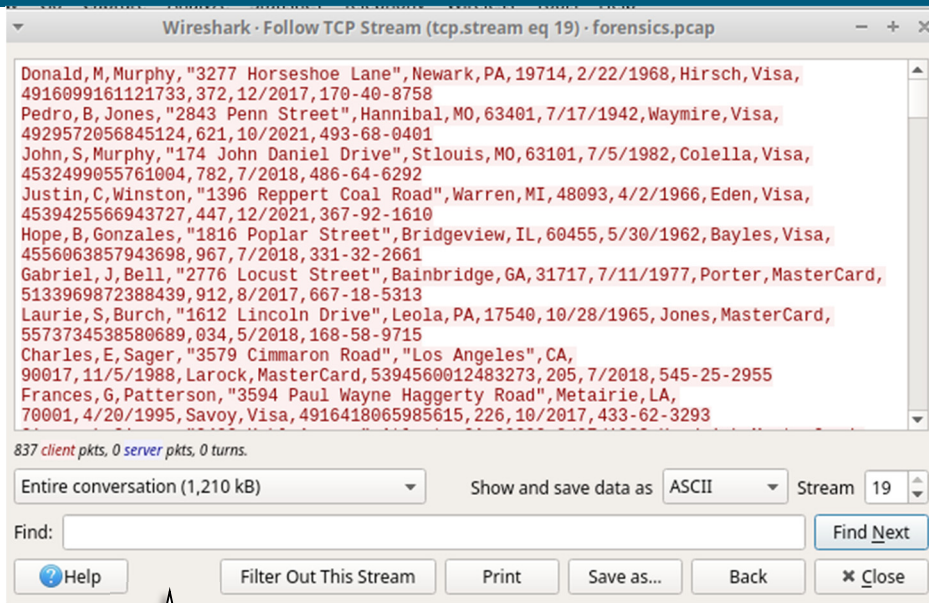
The expected behavior of HTTP traffic is a shorter request followed by a longer response when there is some data other than HTTP status code and headers returned. The above session breaks that rule, so it must be inspected.

We select the **Follow Stream** button at the bottom to examine the suspicious port 80 connection.

✦ To see the output, enter the following on the command line:

wireshark forensics.pcap

Say Goodbye to PCI!



If you called the CIO right now, would she have any questions?

(You should probably call and give an interim report right now.)

Why did we do that?

The news is definitely not good! We see what appears to be *Payment Card Industry (PCI)* records leaving the network. The attacker is exfiltrating the "crown jewels" of the network. How did this happen?

✦ To see the output, enter the following on the command line:

wireshark forensics.pcap

Analyze→Follow TCP Stream



What Was 173.255.224.88 Doing Before This?

No.	Source	Destination	Protocol	Src Port	Dst Port	Info
243	14.4.4.59	173.255.224.88	TCP	80	1492	[TCP Retransmission] 80 → 1492 [ACK] Seq=21001 Ack=41
244	173.255.224.88	14.4.4.59	TCP	1492	80	1492 → 80 [ACK] Seq=413 Ack=21001 Win=64400 Len=0 SLE
245	173.255.224.88	14.4.4.59	TCP	1492	80	1492 → 80 [ACK] Seq=413 Ack=26653 Win=58749 Len=0
246	173.255.224.88	14.4.4.59	TCP	1492	80	[TCP Window Update] 1492 → 80 [ACK] Seq=413 Ack=26653
247	173.255.224.88	14.4.4.59	TCP	1492	80	1492 → 80 [FIN, ACK] Seq=413 Ack=26653 Win=64400 Len=
248	14.4.4.59	173.255.224.88	TCP	80	1492	80 → 1492 [ACK] Seq=26653 Ack=414 Win=6432 Len=0
402	173.255.224.88	55.9.19.52	TCP	43100	80	43100 → 80 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK
403	173.255.224.88	55.9.19.52	TCP	43100	80	43100 → 80 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=15

Exfiltration starts in packet 402; what's in the session before that?

```
GET /img/pfqa.php HTTP/1.1
Accept: image/gif, image/x-bitmap, image/jpeg, application/x-shockwave-flash,
application/msword, application/vnd.ms-powerpoint, application/vnd.ms-excel, */*
Referer: http://wickedsecurity.com/
Accept-Language: en-us
UA-CPU: x86
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)
Host: wickedsecurity.com
Connection: Keep-Alive

HTTP/1.1 200 OK
Date: Mon, 22 Jun 2009 18:18:30 GMT
Server: Apache/2.2.3 (CentOS)
X-Powered-By: PHP/5.1.6
Accept-Ranges: bytes
Content-Length: 26397
Content-Disposition: inline; filename=641.pdf
```

Request for oddly named script

The script sends us a PDF file.

SANS



forensics.pcap

SEC503: Intrusion Detection In-Depth 42

Logically, let's view all the traffic involving the victim IP address using the following Wireshark display filter:

```
ip.addr == 173.255.224.88
```

There is a GET request for **/img/pfqa.php** from host 173.255.224.88. We analyze this by right-clicking on any records in the session and selecting **Follow the TCP Stream**.



To see the output, enter the following on the command line:

```
wireshark forensics.pcap
```

What's in the PDF File?

The image shows a Wireshark packet capture window titled "Wireshark · Follow TCP Stream (tcp.stream eq 7) · forensics.pcap". The main pane displays a JavaScript stream starting with:


```
<<
/S /JavaScript
/JS (
var y = eval;
var s = "17 125 104 121 39 119 104 128 115 118 104 107 39 68 39 124 117 108 122 106
104 119 108 47 41 44 124 55 72 55 72 44 124 55 72 55 72 44 124 55 72 55 72 41 50 41
44 124 76 56 75 64 44 124 58 59 75 64 44 124 60 63 57 59 44 124 60 63 60 63 44 124 58
58 60 63 44 124 73 58 75 73 44 124 55 58 56 74 44 124 58 56 74 58 44 124 61 61 74 64
44 124 76 64 63 56 44 124 77 72 61 60 44 124 58 55 63 55 44 124 59 55 57 56 44 124 77
72 76 57 44 124 56 62 74 64 44 124 57 56 57 57 44 124 59 64 57 56 44 124 55 56 57 56
44 124 57 56 57 56 44 124 57 56 59 73 44 124 77 56 75 76 44 124 57 56 64 63 44 124 57
56 58 56 44 124 72 72 57 56 44 124 74 72 75 64 44 124 62 77 57 59 44 124 63 60 75 57
44 124 77 56 75 76 44 124 75 62 74 64 44 124 75 76 75 76 44 124 74 64 75 76 44 124 57
57 56 74 44 124 57 56 57 56 44 124 75 64 72 72 44 124 56 64 74 64 44 124 57 56 57 56
44 124 74 64 57 56 44 124 57 55 61 74 44 124 57 56 57 56 44 124 61 62 74 64 44 124 57
56 57 56 44 124 74 64 57 56 44 124 57 57 77 72 44 124 57 56 57 56 44 124 75 64 72 72
44 124 55 58 74 64 44 124 57 56 57 56 44 124 74 64 57 56 44 124 57 55 61 60 44 124 57
56 57 56 44 124 56 56 74 64 44 124 57 56 57 56 44 124 74 64 57 56 44 124 57 57 72 63
44 124 57 56 57 56 44 124 75 64 72 72 44 124 57 75 74 64 44 124 57 56 57 56 44 124 74
```

 The bottom pane shows packet statistics: "Packet 207. 1 client pkt, 20 server pkts, 1 turn. Click to select." Below this are controls for "Entire conversation (4,612 bytes)", "Show and save data as ASCII", and "Stream 7". At the bottom are buttons for "Find:", "Find Next", "Help", "Filter Out This Stream", "Print", "Save as...", "Back", and "Close".

Why is there JavaScript in the PDF file?

It appears to create a string of bytes that are later evaluated as code.

The Stream Content reassembly has been separated into two different displays because the actual display and content are quite long, and there are some particular disconnected pieces that need to be highlighted.

The first item of interest is that the server's HTTP **Content-Type** header indicates that the format of the data is **application/pdf**. However, if you scroll down in the **Stream Content**, you see that JavaScript is sent. It seems reasonable to conclude that some type of malware has been downloaded that enabled access to the victim host from which to exfiltrate records.

✦ To see the output, enter the following on the command line:

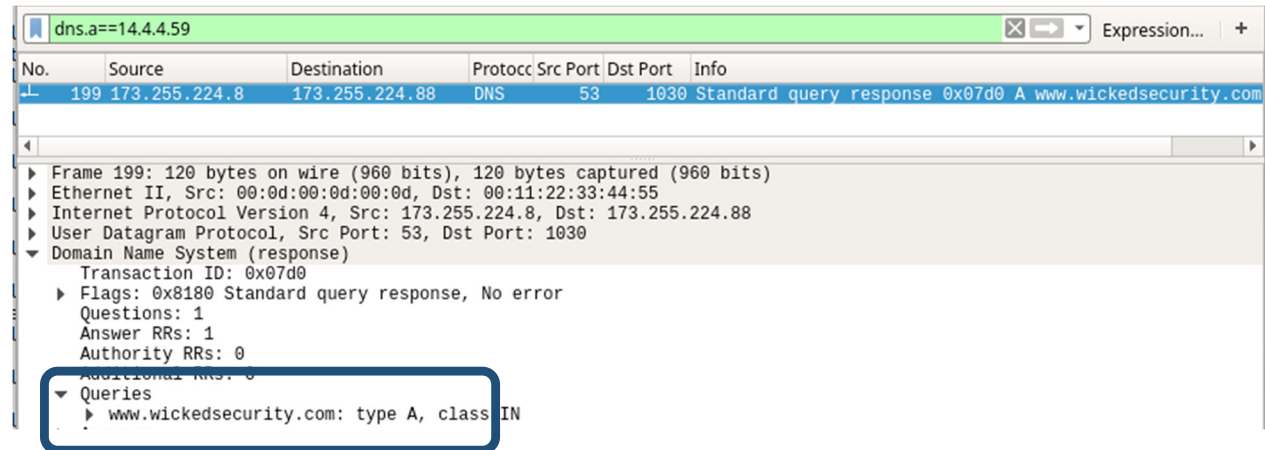
wireshark forensics.pcap

Analyze → Follow TCP Stream

SANS

Great. But we're still not ready to call this done. What we know is that there was an exfiltration. Additionally, we feel pretty confident that the malware that was accessed immediately before the exfiltration is likely responsible. There's still a great deal that we don't know.

DNS Query from Victim IP 173.255.224.88?



If we went to a website, we likely followed a link.
 If we followed a link, that link likely had a URL with a host name in it.
 Why did we go to the website to download the malware? Let's ask DNS!



Since we know that the compromised host visited a website to download the malware, we need to figure out *why* it went to that site. The first link in the chain that solves that puzzle is figuring out how the host knew to go to that IP address. Since it's a website, it seems completely reasonable that a link was clicked on. It also seems reasonable that the link would have contained a URL with a host and domain name. Can we find a related DNS query?

To see if such a query exists, we can look for DNS address responses (DNS A records) where the returned address is 14.4.4.59. To search for these, we can use a display filter of **dns.a==14.4.4.59**.

Entering in this filter, we immediately find a single response. That response reveals that the name searched for was **www.wickedsecurity.com**.



To see the output, enter the following on the command line:

wireshark forensics.pcap

Why Did Our Host Look Up wickedsecurity.com?

No.	Source	Destination	Protocol	Src Port	Dst Port	Info
50	65.55.111.101	173.255.224.66	SMTP	57036	25	C: DATA fragment, 1464 bytes
51	65.55.111.101	173.255.224.66	SMTP	57036	25	from: Jesse B <loser@hotmail.com>, subject: Required Em
73	65.55.111.101	173.255.224.66	SMTP	50182	25	C: DATA fragment, 1464 bytes
74	65.55.111.101	173.255.224.66	SMTP	50182	25	from: Jesse B <loser@hotmail.com>, subject: Required Em
96	65.55.111.101	173.255.224.66	SMTP	35654	25	C: DATA fragment, 1463 bytes
97	65.55.111.101	173.255.224.66	SMTP	35654	25	from: Jesse B <loser@hotmail.com>, subject: Required Em
119	65.55.111.101	173.255.224.66	SMTP	21893	25	C: DATA fragment, 1462 bytes
120	65.55.111.101	173.255.224.66	SMTP	21893	25	from: Jesse B <loser@hotmail.com>, subject: Required Em
142	65.55.111.101	173.255.224.66	SMTP	3585	25	C: DATA fragment, 1463 bytes
143	65.55.111.101	173.255.224.66	SMTP	3585	25	from: Jesse B <loser@hotmail.com>, subject: Required Em
165	65.55.111.101	173.255.224.66	SMTP	40829	25	C: DATA fragment, 1463 bytes
166	65.55.111.101	173.255.224.66	SMTP	40829	25	from: Jesse B <loser@hotmail.com>, subject: Required Em
188	65.55.111.101	173.255.224.66	SMTP	28444	25	C: DATA fragment, 1463 bytes
189	65.55.111.101	173.255.224.66	SMTP	28444	25	from: Jesse B <loser@hotmail.com>, subject: Required Em



Presumably, victim host 173.255.224.88 issued a DNS request for **www.wickedsecurity.com** because it was referenced somewhere in another session. We don't know if this is a TCP or UDP session; therefore, we can search both by looking for the content in IP using the following Wireshark display filter:

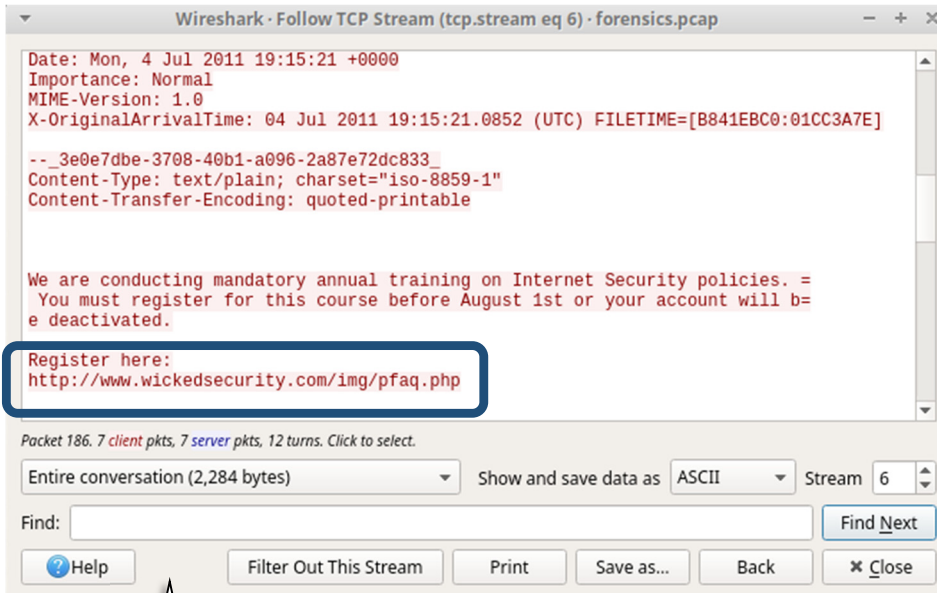
ip contains "www.wickedsecurity.com"



To see the output, enter the following on the command line:

wireshark forensics.pcap

Phishing Email



We now know not only *what* happened, but *why* and *when*.

Could we also identify other users who received the same email?

We discover that the content of **www.wickedsecurity.com** appears in several different SMTP conversations with email directed to different users. A user from victim IP 173.255.224.88 must have read the SMTP message at some point, but the session was not captured. The user clicked on the link, malware was downloaded, and data was exfiltrated.

There are other users to whom the email was sent; therefore, it would be prudent for the site to block outbound access to the malicious web servers. Also, we saw three TCP port 80 sessions when looking at the TCP conversations. The IP address 173.255.224.59 connected to port 80 of host 14.4.4.55 and downloaded the same malware. Yet it appears that it did not perform DNS resolution and makes no other outbound connections. This host needs to be examined for the malware, and it should be taken off the network.

✦ To see the output, enter the following on the command line:

wireshark forensics.pcap



Before the Lab

Wireshark has many more features than we have covered so far.

- You will use some of these features later in this book.
- Some of them are only deeply examined in the labs, so don't skip them!

In this next lab, you will work with tshark:

- Command-line interface to Wireshark
- Supports pretty much everything that you can do in Wireshark
 - Extracting files with tshark is very difficult, but it can be done.
- You can use display filters at the command line.
- You can choose which fields you want to see displayed!

Know that there are many more features in Wireshark than we have covered. While we will explore some of these features in later sections, we strongly encourage you to explore this tool! Also, there are some very valuable aspects of this tool that we will only examine deeply in the associated lab exercises, so be sure not to skip any!

One of these features is the tshark tool, which is a command-line version of Wireshark. Suppose you want to follow TCP sequence numbers in a series of packets. Comparing this in Wireshark would be time consuming; tshark's command-line output is more appropriate for a task like this.

Tshark can employ the same display filters that Wireshark uses. The **-Y** command-line switch identifies that you want to use a display filter. You must know the exact display filter because tshark, unlike Wireshark, offers no automated assistance in forming the display filter expression. For example, not only can you specify something like **ip.src == 10.3.8.108**, but you can also use application layer display filters like **http.request.method == GET** to look for all HTTP GET requests.

You will have the opportunity to try some of this out in the upcoming lab.

Wireshark III Summary

Export web objects for deeper inspection.

Carve out base64 MIME-encoded SMTP attachments.

Excellent for investigating traffic in many different ways: statistical overview, stream reassembly, progression of events

Many other features, such as command-line tshark

This section wraps up our extensive coverage of Wireshark. Hopefully, you see the power and utility of Wireshark given the material covered in these books so far. There is far more that this tool can do, and we will experiment with it more in the bootcamp at the end of this volume. Continue exploring and experimenting with this tool!

In this section, you learned about extracting web objects from HTTP traffic and carving out files, and later examining base64 MIME-encoded SMTP attachments. This permits you to inspect more content than just stream reassembly.

The particular sample exploit traffic that you followed showed how to approach the use of Wireshark to logically allow you to progress from a statistical overview to a more refined inspection of different aspects of the traffic.

Wireshark has many more features than we've covered in all the sections devoted to it in this course. The command-line version of Wireshark is tshark.

Wireshark Part III Exercises

Workbook exercises "Wireshark Part III"

This page intentionally left blank.

• An immersion class in speaking packets, protocols, and hex • Ethernet • IPv4 & IPv6 • TCP, UDP, & ICMP • Wireshark & tcpdump fundamentals, filtering, & investigation	Section 2: Signature-Based Detection • Snappy for building tools and simulating signatures & behaviors • Snort & Suricata • Researching common application protocols	• Architecting for massive packet capture & analysis • Overcoming the limitations of signature-based tools • Basic Zeek usage and logs • Threat modelling • Advanced correlation using Zeek	• Guided scenarios to reinforce all skills • Elaboration on effective deployment, collection, and analysis at large scale • Moving beyond alerts to data, driven analysis, & machine learning	• Challenge questions guide/follow expert analysts' progression through real-world data
---	---	---	---	---

- Packet Crafting for IDS/IPS
- Wireshark Part III
- Application Protocols with Snort and Suricata

Signature-Based Detection

Introduction to Snort

Writing Rules

Content Options

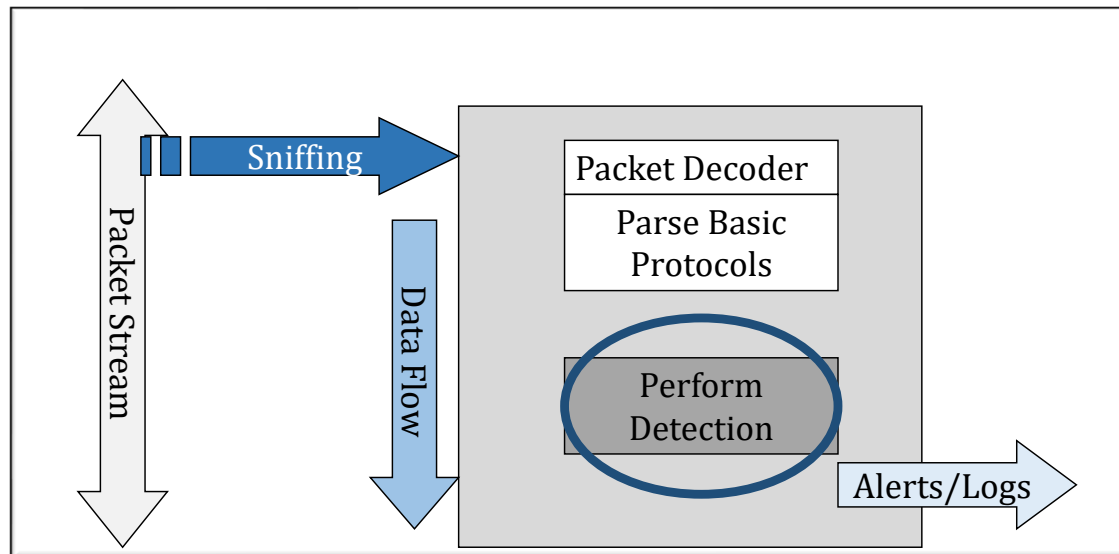
Streams

HTTP and HTTP Rules

DNS and Byte Test

The time has come to start to put what you have learned into action with a real IDS. The primary focus for the remainder of this book is the effective use of Snort and other Snort-like IDS tools. Snort is and has been the most popular IDS/IPS solution and is the most widely installed IDS/IPS on the planet, either as the free, open-source version or as the commercial version. The free version is available from <http://snort.org>. The commercial version was known as Sourcefire, but Sourcefire was acquired by Cisco in 2013 for more than \$2 billion. Sourcefire has been rebranded as FirePOWER, though you can find the Snort engine embedded in other products. For example, the Symantec endpoint protection offering has an embedded Snort instance performing real-time detection. You will also find it embedded in many next-generation firewalls.

What Does an IDS Do?



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

The phases that occur before and after detection are important and must function well for the IDS/IPS to do its job, but most of the focus for us as analysts is the detection part. There is an output phase that follows the detection, where alerts, logs, and notifications are somehow processed for viewing or further analysis. Not to diminish any of the other phases, but the significance for us is the proficiency of detection and the interface the solution offers us to facilitate designation of what is considered noteworthy.

The Analyst's Role in Detection

Best detection offers the analyst:

- Visibility into the detection process
- The capability or "language" that permits you to define what is considered malicious or noteworthy:
 - Snort rules
 - Zeek scripting

Any credible IDS/IPS solution will:

- Allow user to view, alter, and create new rules, scripts, or detection methodology
- Offer the opportunity to see the traffic that caused an alert or notification

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

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

In our discussion of Snort and Zeek, we discover that the fundamental language Snort uses is rules, whereas Zeek uses scripts in a language written expressly for it. Snort rules provide us a language, allowing us to designate packet characteristics of interest. Basically, Snort offers a more user-friendly (but limited) language and does most of the processing of the rules under the hood. On the other hand, Zeek exposes an entire programming language to us. Although this permits far more manipulation of the detection process, it is more difficult to learn because you must learn the intricacies of the language itself and figure out how it can accomplish what you want.

Our opinion is that any credible IDS/IPS solution must offer transparency to the user in the form of visibility into the detection process, whether that is rules, scripts, or any other methodology. Also, the solution must offer the opportunity for the analyst to see the packet(s) associated with an alert of some type of notification of noteworthy activity. A failure to present either of these options returns the user to the "check engine light" alert analogy, with some kind of red blinky light indication of potential trouble.

Failure to present the analyst with this requisite data may not be deception, but there is some amount of secrecy or arrogance in the idea that the user does not need access because the solution does it all or the user is incapable of understanding more than a blinky light.

Why We Matter

Oh no! This looks bad.

```
[1:57756:2] "MALWARE-CNC DNS Fast Flux attempt" [**] [Classification: A Network Trojan was
detected] [Priority: 1] {UDP} 8.8.8.8:53 -> 192.168.2.239:52224
[1:57756:2] "MALWARE-CNC DNS Fast Flux attempt" [**] [Classification: A Network Trojan was
detected] [Priority: 1] {UDP} 8.8.8.8:53 -> 192.168.2.244:43140
[1:57756:2] "MALWARE-CNC DNS Fast Flux attempt" [**] [Classification: A Network Trojan was
detected] [Priority: 1] {UDP} 8.8.8.8:53 -> 192.168.2.240:45308
[1:57756:2] "MALWARE-CNC DNS Fast Flux attempt" [**] [Classification: A Network Trojan was
detected] [Priority: 1] {UDP} 8.8.8.8:53 -> 192.168.2.240:45308
```

What's in those packets??

```
15:08:26.461953 IP 8.8.8.8.53 > 192.168.2.239.52224: 12572 7/0/0 CNAME
outlook.ha.office365.com., CNAME outlook.ms-acdc.office.com., CNAME MNZ-efz.ms-
acdc.office.com., A 52.96.32.178, A 52.96.109.178, A 52.96.69.2, A 52.96.181.34 (187)

0x0000: 4500 00d7 cddb 0000 7911 9f93 0808 0808 E.....y.....
0x0010: c0a8 02ef 0035 cc00 00c3 50d6 311c 8180 .....5....P.1...
0x0020: 0001 0007 0000 0000 076f 7574 6c6f 6f6b .....outlook
0x0030: 096f 6666 6963 6533 3635 0363 6f6d 0000 .office365.com..
```

Installing and operating a network monitoring tool is not difficult. Within minutes you can have a system that is generating alerts to your SIEM or another alert console. However, simply generating alerts does not provide value to the organization. Consider the alerts in the slide.

If you were to see these alerts, how would you interpret them? The message seems to involve some kind of malware command and control (CNC or C2) activity. The classification string seems unambiguous: "A Network Trojan was detected." Would those alerts trigger your organization to scramble an incident response?

Let's look at what's happening a bit more calmly before we call in the incident handling brigade. We will discuss DNS at some depth later in this book. For now, take note that these are DNS response packets entering the organization that are generating the alerts. Looking at these responses, we can see that the resolution is for something called "outlook.ha.office365.com." This appears to be a DNS response providing multiple addresses for a high availability (HA) load balanced Outlook cluster hosted by Microsoft at office365.com. Could this be malware? Sure. However, in this case, it almost certainly isn't. Why, then, does the monitoring tool give us these really serious sounding alerts?

Frankly, signature-based tools are just not that smart. There is a rule in the rule feed that looks for DNS responses with short Time to Live values (DNS TTL, not IP TTL; more in the DNS section) for the same host or domain. When it sees these, it fires an alert. However, what we are seeing here is *completely normal activity for Office 365!* How many times will your organization overreact to alerts from your security tools before you begin to ignore the tool?

To complete the thought here, though, think about how we know this is a false positive. It is our knowledge of how networks and protocols function coupled with our ability to deeply understand what is happening in the packets that allows us to diagnose this activity as innocuous.

The Analyst's Means of Customizing Detection

Initial configuration

- Customization for a specific IDS/IPS, depending on sensor location and purpose
 - Enable processing that occurs before inspection.
 - Enable or disable specific protocol processing.
 - Include rules or scripts.
 - Include/remove specific rules files.
 - Add unique local site inspection.

Ongoing configuration

- Refine above configurations.

Update software and add new rules or scripts.

Backend processing, including reporting

As an analyst, you will need to customize an IDS/IPS for your site's requirements. Undertaking and successfully performing customization can be the difference between well-tuned monitoring and generic monitoring with an overabundance of false positives and/or the introduction of false negatives as the consequences. It is possible that the existence of multiple sensors will require different configurations or customizations. Take the case of a focused IPS that monitors traffic into and out of an intranet with customer records. It requires a different set of initial configuration changes than an IDS that sits just inside the firewall.

Packet(s) can be processed before they undergo inspection. Snort, for instance, is able to reassemble fragments into the pre-fragmented form and reassemble streams into a session of related packets. Customization of these processes can encompass target-based detection that applies a different reassembly policy for different operating systems for anomalous fragments and packets (say, where there is an overlap of content that isn't due to network issues). This can protect against evasion and insertion attacks when done properly.

Some Snort preprocessors decode different protocols, according to specifications such as HTTP. It is wasteful to include any preprocessors that decode protocols that are not used in your network. It is advantageous to tune the enabled preprocessors for your site, too.

Both Snort and Zeek have default rules and scripts that accompany them or are available with separate downloads. The files that contain them are included in their respective configuration files. It is possible that some of these rules pertain to protocols that you do not use; therefore, they can be commented out to make the processing more efficient. And it is possible that there are no rules or scripts that perform some site-specific inspection that would benefit from rules and scripts that you write. All the tasks that have been discussed may have to be refined and reiterated intermittently. Finally, you might be responsible for setting up reporting and backend processing that provides the best insight possible into the activity that has occurred.

What We Will Cover

- **Creating detection capability from CVE or other reports describing vulnerabilities**
 - This is something you should be doing frequently.
 - This is something you should be able to do well.
- **Performing threat analysis to describe and detect unusual behaviors that could be indicators of compromise or events of interest**
 - This is something you should be doing, but probably are not doing.
 - This is something you should become adept at doing.

Now that you know why we are teaching the material we have covered and will cover, let's set the expectations of what will be taught. A savvy analyst must be aware of newly announced vulnerabilities, the impact they may have on the monitored site, and the need to write new rules or scripts if none exist. This is applicable also if there is a time gap between the announcement of the issue and the release of new rules or scripts.

You need to stay informed about these new vulnerabilities, and it is beneficial to become proficient at writing rules or scripts, even at a basic level, if coverage is lacking before detection becomes available. Patches may be issued that can mitigate the impact, but patching may not be done immediately, if at all.

Sometimes a single rule or script is not enough to detect a particular behavior that reflects a compromise or an event of interest. For instance, suppose a user receives a phishing email to click on a link to a malicious site. How could that be detected by decomposing the process into smaller behaviors? First, a link would need to be discovered in SMTP data, a DNS lookup would most likely be required to convert a hostname into an IP address, and then traffic would be seen to the malicious site.

This is just one example of behavior-based detection. There may be behaviors specific to your site that can be translated into detection. Say, for instance, there is a database of customer data. A large volume of database records transferred to unexpected places could be anomalous site-specific behavior. Chances are that you have not thought in these terms, either because it wasn't obvious or you did not have the means to implement your analysis. You will have the skills and tools to do this after completing the course.

Another IDS/IPS: Suricata

- Originally created as a fork of Snort
 - Rules languages of Snort 2 and Suricata are very similar.
- Multithreaded capability:
 - Suricata has supported this for many years, Snort 3 just added in 2020
- Protocol detection:
 - Suricata detects many protocols via recognition of protocol standards.
 - Snort 3 can do this as well.
 - Snort 2 detects protocols such as HTTP via port assignments in rules.

Right now, all embedded Snorts are version 2!

Although we will not cover it in any detail, we want to mention Suricata, another popular open-source IDS/IPS solution. Originally, Suricata was created as a fork of Snort due to user frustration over inability to contribute rules, code, and the direction of the Snort project. The biggest issue that the Suricata team wanted to address was adding multithreaded detection capabilities to the Snort code base. Snort was and still is open-source; however, the company at the time, Sourcefire, offered a commercial version of the product that was based on the open-source version of Snort.

Sourcefire was reluctant to accept user contributions, relying instead on the talented developers and rule contributions of the company's employees. The company was fearful that user-submitted contributions might harm performance or, worse yet, introduce vulnerabilities into Snort. Users who wanted to contribute their own ideas and code decided to use Snort for base concepts to create their own version of an open-source IDS/IPS solution known as Suricata.

Suricata began to grow and gain more recognition, especially when they added the capability of multi-threading, allowing many different tasks to be processed at once with an attendant improvement of speed and efficiency. Snort had dangled version 3.0 for many years as its multithreaded solution. Snort 3 finally exited beta in 2020, over a decade from when it was first teased by the Snort team.

Another major difference between Snort 2 and Suricata is Suricata's capability to be protocol-aware, such as for HTTP, by examining the payload for protocol standards. For instance, the presence of a URI-like format in an HTTP request along with HTTP headers might signal that HTTP is in use. Snort 2 relies on ports to define a protocol, such as ports 80 and 8080 for HTTP. The user can alter the Snort configuration file to indicate the HTTP ports in use on the monitoring site, but it can become cumbersome to maintain all supported HTTP ports on the monitored network. Also, Snort 2 would not be able to detect HTTP traffic over a nonstandard port, one that supports HTTP tunnel traffic.

It's worth pointing out that, while Snort 3 is multithreaded and has added some wonderful capabilities, the deployed products that have the Snort engine embedded in them are all, as of 2021, still running the Snort 2 engine with all of its limitations.

Analysis Capabilities and Limitations

Snort does have limitations.

- Snort rules may lack the fidelity to scrutinize the packets/stream in the granularity that you need.
- You can identify only what rules language options can detect.
- Snort rules lack the capability to perform correlation across time and systems.
- Snort lacks some important application decode capabilities.
- Snort 2 and 3 seem to have a hard limit of five alerts per packet.

You need to be aware what your blind spots are!

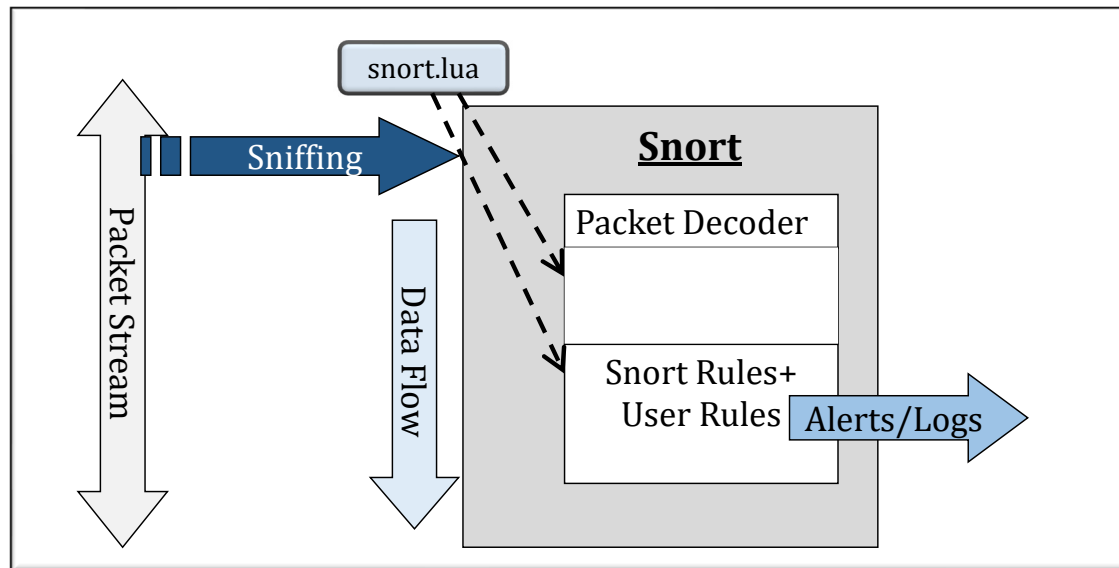


As you will soon learn, Snort offers a rules language that is quite comprehensive for individual and reassembled packet inspection. But there are limitations to the Snort rule language. What if the rule language does not offer a capability to do the type of inspection you want or the type of granular inspection you want? You may end up with an imprecise rule that may trigger false positives or negatives. What if you need to perform correlation across many different events, using data from one event to seed the next? That's just not something that signature-based tools can do, including Snort. What if you want to examine something in a protocol header that the rules language doesn't expose? Sorry, you can't look at it.

Snort is able to reassemble a given session from all the component packets. Yet it cannot perform any inspection between different sessions. The phishing attack was comprised of three phases—the SMTP traffic with a link, the DNS resolution of the hostname in the link, and the subsequent connection to the resolved IP address. Evaluation of these three individual events as related activity might find malicious activity that might otherwise not be found. Plus, the capability to have associated indicators can potentially reduce false positives.

A serious flaw that comes into play in our discussion of evading detection is the apparent hard limit on the number of alerts that can be generated from any single packet. Snort 2 has a hard limit of five regardless of what the documentation claims. From our testing, it seems that Snort 3 has this same limit even though the documentation indicates the maximum should be 32. However, even if the maximum is 32, this can still represent a serious evasion/insertion opportunity. We will talk about this in depth at the end of Book 4.

Snort Process Flow



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

A configuration file named **snort.lua** is included in the Snort install. It tells Snort which preprocessors to enable and which rules to include. Certain preprocessors might be needed at a particular site along with specific rulesets that are pertinent for the environment. The user should tune **snort.lua** for the site's environment.

How to Create Site- or Location-Specific Customized Detection

/etc/snort/snort.lua controls everything in Snort.

- Enable/disable preprocessors and configuring preprocessor parameters.
- Assign values to variables that identify monitored network, servers, ports
- Inclusion of rules files

Rules

- Choose which rule groups to use.
- Suppress or modify rules.
- Create your own rules.

```
-----
dir = '/sec503/Exercises/Day3/etc-snort/'
if ( not dir ) then
    dir = '.'
end
dofile(dir .. '/snort_defaults.lua')

HOME_NET = 'any'
EXTERNAL_NET = 'any'
DNS_SERVERS = HOME_NET
SMTP_SERVERS = HOME_NET
HTTP_SERVERS = HOME_NET
SQL_SERVERS = HOME_NET
TELNET_SERVERS = HOME_NET
SSH_SERVERS = HOME_NET
FTP_SERVERS = HOME_NET
SIP_SERVERS = HOME_NET
HTTP_PORTS = [ 36 80 81 82 83 84 85 86 87 88 89 90 311 383 555 591 593 631
801 808 818 901 972 1158 1220 1414 1533 1741 1812 1830 1942 2231 2301
2381 2578 2809 2980 3029 3037 3057 3128 3443 3702 4000 4343 4848 5000
5117 5250 5450 5600 5814 6080 6173 6988 7000 7001 7005 7071 7144 7145
7510 7770 7777 7778 7779 8000 8001 8008 8014 8015 8020 8028 8040 8080
8081 8082 8085 8088 8090 8118 8123 8180 8181 8182 8222 8243 8280 8300
8333 8344 8400 8443 8500 8509 8787 8800 8888 8899 8983 9000 9002 9060
9080 9090 9091 9111 9290 9443 9447 9710 9788 9999 10000 11371 12601 13014
15489 19980 29991 33300 34412 34443 34444 40007 41080 44449 50000 50002
51423 53331 55252 55555 56712 ]

SHELLCODE_PORTS = '!80'
```

We covered the topic and importance of the analyst's role in optimizing the accuracy of any IDS/IPS monitoring. More specifically, **snort.conf** is the primary configuration file where preprocessors are enabled, disabled, and assigned parameter values that reflect your expected network activity. This is where some universal and important variable values are defined. Snort rules often differentiate between IP addresses that belong to the monitored network and those that do not. Therefore, these have to be assigned respective values of **HOME_NET** and **EXTERNAL_NET**. If these are not defined or defined properly, it is possible that false negatives and false positives will be generated.

There are other variables that define the IP addresses of various servers. For instance, you can identify the HTTP servers in the monitored network by assigning **HTTP_SERVERS** the IP addresses of those servers. You can also define the listening ports on web servers with the port variable **HTTP_PORTS**. A rule that references these variables restricts the traffic that is examined, with the likely potential of increasing efficiency when the rule is compared with the traffic. As we will soon learn, search optimization can be realized when rule conditions eliminate packets or streams early in the matching process. It is good to note that the use of the **HTTP_PORTS** variable is really a legacy configuration at this point. Snort 3 performs dynamic protocol detection and will detect HTTP running on other ports. However, until you are sure that all of your rules and rule feeds are fully Snort 3 compatible, it would probably be best to retain some of these legacy configuration options. Otherwise, those rules will just break!

Rules can be included or excluded in two ways. The **snort.lua** file includes rule files that tend to group rules by function or software. The retrieval and configuration of which rules are active is typically handled by an external piece of software like PulledPork.

Your team may wish to write rules or change existing rules if you find something is not covered. The file **local.rules** is the location for user-added rules.

Import Snort Configuration Files

- `snort.lua`
 - The main Snort configuration, this config "includes" all the others using `dofile()`.
 - `snort2lua` tends to put everything into the `snort.lua` file!
- `snort_defaults.lua`
 - Configures many default options
 - Includes things like external references, default classification categories, and other general configuration
- `threshold.lua`
 - Used to configure rate limiting for alerts, in addition to suppression of rules or based on addresses
- All the rules/signatures

The Snort configuration is just one huge text file. Under Snort 2, we could use the **include** directive to include various files. Since Snort 3 now uses Lua as language for parsing the configuration, we can include other Lua files with the **dofile()** function. Both of these approaches allow the Snort configuration to be broken into multiple modular components, which are then combined at runtime. **snort.lua** is the main configuration that is loaded, and it typically imports all these other components, in addition to configuring default logging options, preprocessor settings, and variables. Be aware that if you are migrating from a Snort 2 to a Snort 3 install, the **snort2lua** command can be used to perform the conversion. It will, however, tend to smush everything into the **snort.lua** file rather than maintaining the modular files.

We will see that the rules language allows you to attach a classification and a priority to any given rule. To simplify things, you can use the **adjust**, **remove**, or **add** categories using either a separate **classification.lua** file or within the **snort_defaults.lua** file. This is great, because every rule in the public rule feeds includes a classification. This allows you to customize how important you believe the different classifications are, immediately making the alerts more relevant for you.

Most rules in the public feeds also include metadata for the source information for the attack or vulnerability. These references are configured in **snort_defaults.lua** as well. You can modify this list, adding additional references, which can then immediately be used in rules to allow easier research.

The **threshold.lua** file is used for several things. It allows you to create limits, or thresholds, for how many times any given rule will fire an alert in a given time frame. This can prevent you from being overwhelmed. This file also allows you to add configurations that will suppress specific rules from ever firing an alert or perhaps prevent alerts from being generated from a particular host or network block. This is especially helpful when you have a few noisy rules in an otherwise useful rule feed. Rather than discard the entire feed, you can configure suppression of the alerts that you don't care to see. While we can suppress rules here, it is probably more common to do so using the configuration of your rules management tool, which is usually PulledPork.

Finally, we have all of our individual rule files. These files are generally located in **/etc/snort/rules** and are organized into files that group the rules by function. For example, web rules are in one file, while SMTP rules

are in another. This allows you to choose to include or exclude broad classes of rules in your configuration quickly and easily. PulledPork tends not to maintain these rules in separate files, but instead merges them all into a single **snort.rules** file.

Preprocessing

Snort 3 takes a more efficient approach over Snort 2.

- Snort 2 used a "Just in Case"; Snort 3 uses "Just in Time."

Preprocessing activities for Snort 3:

- Decode packets.
- Reassemble TCP streams.
- Detect protocols using protocol inspectors.
 - Normalize and publish data as needed and when detected.

Snort 3 and Snort 2 operate in fundamentally different ways. Snort 2 took a "Just in Case" approach. For example, if the HTTP preprocessor was installed and configured, it would attempt to preprocess all HTTP streams seen on configured HTTP ports, extracting cookies, URIs, etc. It would perform all of these actions even if there were no rules that made use of some (or even any) of those pieces of data.

Snort 3, on the other hand, uses a "Just in Time" approach. Not only does Snort 3 dynamically detect a protocol with a protocol inspector, such as http, rather than requiring you to define all the ports where http data will be present, but it will only normalize and/or decode the data for the various fields if there is something that is attempting to consume that data. If there are no rules or anything else that examines the URIs in HTTP streams, for example, then the URIs will *never be extracted*. This means that Snort 3 only does work that is required (and it only does that work once) rather than doing a lot of work up front "just in case" something wanted the result later.

During the preprocessing phase, Snort 3 decodes the captured (or read-in) packets using protocol decoders (or "codecs"). The stream reassemblers also go to work behind the scenes to reassemble the TCP and UDP streams that are passing by. Finally, protocol inspectors are triggered as application layer protocols are detected. These then make available, or publish, various pieces of information or data within the application layer that can be consumed by other parts of Snort or within rules.

Running Snort

To run snort, we need a few basics:

- A configuration file
- Packets
- A destination for our alerts

In the first lab, you will explore some of this:

- When logging in production, you might use `unified2`.
 - Unified embeds the alert and the packets into one file.
 - We can view the alert data using `u2spewfoo`.
 - We can extract the packets using `u2boat`.
- For development, we can define `alert_fast` and `log_pcap` in `snort.lua`.

In the first lab, you will experiment with Snort and learn to take a few first steps. Most important is becoming accustomed to some of the common options that are used to run Snort and exploring just a bit of the configuration. One of the things you will run into is learning how to see what alerts are being generated.

Snort provides various output options. We can specify `csv` to obtain CSV-based output, `alert_json` to generate JSON output (which is particularly good for sending to a SIEM), `alert_fast`, which is the most common to use on the command line, and others. One of the commonly configured output options is `unified2`. This format will generate alerts with the packets that caused the alert into a unified file. This file can be viewed using the tool `u2spewfoo`, which will show you the alert text along with a hex dump of the associated packets. You can extract the packets into pcap files using the `u2boat` command.

For working in development, we can run Snort and tell it to generate "fast" alerts to the console using `'-A alert_fast'`. This is great for quick testing. However, when we are trying to debug a rule, we will often want to see the exact packets that generated the alerts. While we can use unified2 format and pull the packets apart from the alerts, you may find this inconvenient.

A simpler approach is to add some logging configuration to the `snort.lua` file in your development environment. To do so, define `alert_fast = {}` and `log_pcap = {}` within your `snort.lua`, and you will now have fast console alerts with pcap log files in the current directory!

Running Snort and Snort Output Exercises

Workbook exercises "Running Snort and Snort Output"

This page intentionally left blank.

• An immersion class in speaking packets, protocols, and hex • Ethernet • IPv4 & IPv6 • TCP, UDP, & ICMP • Wireshark & tcpdump fundamentals, filtering, & investigation	Section 2: Signature-Based Detection • Scapy for building tools and simulating signatures & behaviors • Snort & Suricata • Researching common application protocols	• Architecting for massive packet capture & analysis • Overcoming the limitations of signature-based tools • Basic Zeek usage and logs • Threat modelling • Advanced correlation using Zeek	Section 3: Machine Learning • Guided scenarios to reinforce all skills • Elaboration on effective deployment, collection, and analysis at large scale • Moving beyond alerts to data, driven analysis, & machine learning	• Challenge questions guide/follow expert analysts' progression through real-world data
---	---	---	---	---

- Packet Crafting for IDS/IPS
- Wireshark Part III
- Application Protocols with Snort and Suricata

Signature-Based Detection

Introduction to Snort

Writing Rules

Content Options

Streams

HTTP and HTTP Rules

DNS and Byte Test

Now that we've gotten our feet wet running Snort and we have some understanding of its output, let's try our hand at writing rules! We're going to take this in slow, careful steps, both explaining the various capabilities and options in Snort and introducing you to good practices for rule development.

Understanding and Writing Snort Rules: Continuous Interaction

Understand/write rules? Doesn't Snort come with a set of rules? Yes, but....

- What if a new zero-day vulnerability/exploit appears, and no Snort rules exist?
 - How do you know if an attacker is targeting your network?
 - How do you know if you aren't already compromised?
- What if a rule alerts and you want to determine whether the alert is a false positive?
- What if you discover through some other means that you thought you were covered by a rule(s), but an alert did not fire?
 - Requires you to examine packet/stream content to understand what the rule alert seeks to match
- What if you have something specific to your site that you want to detect for which there is no existing rule?

Suppose that you have installed Snort and loaded up the default rules that accompany it. You've scheduled periodic updates to download and apply any updated rules. Isn't your site covered and properly monitored if you do this? And why should you learn to understand and write Snort rules if there is a commercial company with a group of experts writing them for you?

This is true; however, the Snort rules may not be released soon enough after a zero-day vulnerability is announced. It's not that the Snort rule writers are slow or dim-witted or aren't able to create new rules for the vulnerability. They must write efficient and accurate rules. This means that the created rule(s) must not overburden processing; a poorly written rule can bring Snort to its knees. Moreover, the rules should be precise enough not to generate false positives, if possible. These requirements, along with understanding the vulnerability and translating it into an efficient, accurate, and focused rule, take time to write and even more time to test with different versions of the commercial and open-source offerings. That is why there may be a time gap between the vulnerability release and the creation of a Snort rule to detect an attacker trying to capitalize on the vulnerability, not just a particular exploit for the vulnerability.

When this situation arises, you can read about the vulnerability and create a rudimentary or sophisticated rule to provide coverage. This permits you to know if you are being targeted, and it can examine any recently saved traffic to determine retrospectively if there has been a successful compromise. Another situation where understanding rules is indispensable is when you suspect Snort may be generating many false positives. This can be diagnosed only if you have an understanding of the rule that triggered the alert, then match the rule conditions against the packet or stream that caused it to fire.

Lastly, a situation or condition may be present for which your site wants detection. Suppose there is some proprietary or intellectual property that can be discovered in payload content. It might make sense to look for the presence of this in outbound traffic over specific or general ports as a sign of exfiltration.

Snort Rules

Snort rules have a simple format that allows great flexibility in **single packet** analysis:

- This may be sufficient to pick up some attacks.
- Multi-packet events/attacks are generally prepared for inspection by preprocessors.
- Snort's stream and fragment reassemblers allow stateful rule-based analysis to occur (more on these later).

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

The fact that you are able to see the rules is a significant benefit. Some commercial IDS/IPS offerings keep the rules hidden so you cannot see what causes alerts to happen. This precludes the ability to distinguish false positives from true alerts if you do not know what the rule does. All you need to do to associate a Snort rule with an alert is examine the *Snort ID (SID)* listed in the output, then search the rules directory for the rule with that same SID.

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

Detection Scenario

Want to write a Snort rule to find a Windows command-line prompt opening following successful exploitation of a flaw using Metasploit

Packet characteristics:

- Any traffic from the monitored network to a host outside the monitored network
- Carried in packet(s) in an established TCP session from the client to the server
- Destined for Metasploit communication channel server port 4444
- Data characteristics:
 - Content of "Microsoft Windows"
 - Content of "Copyright (c)"
 - Content of "Microsoft Corporation"

Here is a scenario that we'll use to create a new Snort rule and define the content and content placement found in the packet that we translate into the components and keywords available in Snort's rule language. Let's say that we've decided we'd like to be alerted whenever a Windows command-line prompt is seen as the result of a Metasploit session. The default Metasploit communication channel uses port 4444, although the attacker can select any port not currently in use on the attacking host. Our restriction of port 4444 usage is not ideal, but it simplifies the creation of our very first rule. Later, if we wanted, we could alter it to look for a wider expanse of ports.

In terms of packet characteristics, the traffic is originating from the monitored network and destined for a host outside the monitored network. We will use the variables **HOME_NET** and **EXTERNAL_NET** to represent those for this rule.

We want Snort to examine payload for the content we will supply in the rule only in the context of an established session after the three-way handshake—more specifically, in the direction from the originating client to the server. We'll examine this part in greater detail in the following section.

Basic Rule Anatomy

Each rule has two parts:

- Rule header
- Rule options

Specific syntax for each part

Every rule must have a header and an option section, but options may be empty.

Rules may be entered in the rules file on multiple lines.

- Some shops prefer to require local rules to be single lines, even if they are very long.
- This allows you to easily see which rules have changed when viewing diffs.

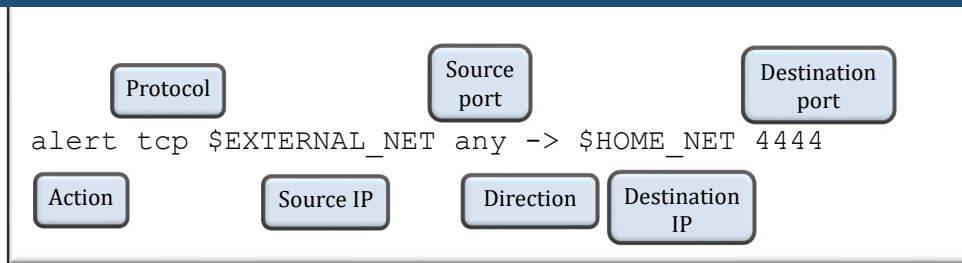
An individual rule is broken into two general parts. The first part (the rule header) defines "who" must be involved for the traffic to be considered by the rule options. The second part (the rule options) defines "what" must be involved. This includes additional packet header traits or the contents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

Both sections are used for most rules. It is possible to create rules with a rule header but an empty options section. This is done so that some action can be taken for the provided hosts and ports. This is usually the case where *pass* action rules are used to ignore traffic between given hosts and ports.

In Snort 2, you could define rules that took multiple lines, but each line would require a backslash continuation character. With Snort 3, rules may cover multiple lines, and no special continuation characters are required.

Even though you can write rules that are split across multiple lines, you may find that some organizations prefer not to use this feature. The reason is that some will manage local rules in a code repository like git. These tools are wonderful for discovering which lines have changed in a source file. If a rule is split across multiple lines and a change is made, finding the change is simple, but additional work would be required to determine which rule has been changed. By requiring that rules are single lines, even if they are quite long, it makes identifying which rules have changed simpler.

Rule Header



Defines who is involved:

- Action: IDS alert, pass, log, activate, dynamic; IPS drop, sdrops, or reject
- Protocol: ip, tcp, udp, icmp
- Source/destination IPs: One or more addresses or networks, can be defined as variables
- Source/destination ports: One or more ports, can be defined as variables
- Direction of traffic: Arrow designates where traffic originates from and is destined

The *rule header* is the first part of each rule and is used to define the rule action, the network protocol, and "who" must be involved.

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

The first field of any rule is the action that should be taken if the rule triggers. Common rule actions are to **alert** and/or **log** for an IDS. This means that the alert will be directed to the screen or a log file. The log file format is one that should be used when Snort is in live detection mode. We will be examining Snort traffic in readback mode in the examples where Snort processes a captured file, and we will opt to direct the alerts to the screen. The next field is the protocol in the traffic; choices are **tcp**, **udp**, **icmp**, and **ip**.

Source and destination IP addresses and ports can be specified in different ways, including variables, single values, and lists of values. And there is a traffic direction indicator that identifies either a unidirectional or bidirectional flow of the expected origin and destination. A simple example would be where you want to be alerted only if the traffic is destined for a host in the monitored network versus alerting if the traffic originates from the monitored network. This restriction can improve efficiency or coverage significantly. The bidirectional case implies that the direction is irrelevant.

The fields that represent IP addresses and ports can have a single value, a range of values, or a list of values, or they can use the designation of **any** to represent any value.

Service Headers



Snort 3 allows us to write rules based on the detected services!

- We still define an action.
- Rather than defining a protocol, we use a service inspector.
 - http, ftp, telnet, ssh, ssl, file...
 - There is no error if you define an inspector that doesn't exist!
- Allows us to apply options to a service rather than hosts and ports!

A major change in Snort 3 is the idea of **service inspectors**. A service inspector is a plugin to Snort that understands how to identify, parse, perform anomaly detection on, and normalize (where appropriate) some specific protocol or service. At least one of these—the **file** service—is more general, allowing you to apply rule options to any identified file transfer, whether it was over FTP, IMAP, HTTP, SMB, or service inspector that exposes file events.

At first, it might not be apparent why this is so important. Under Snort 2, we would have been required to create a rule header indicating the source, destination, and ports involved. If we wanted to analyze HTTP traffic, we would have been limited to specifying the ports where we expect that to occur. Additional configuration would have been required to configure the http preprocessor to decode specific ports as that type of data. Under this regime, if we wanted to write a rule, we would have to specify all the ports every time. While this was simplified by using port variables, Snort 3 makes this approach obsolete.

Now we need to simply specify the service that we are interested in, and Snort works out the rest. It no longer matters which port the traffic was seen on or if our preprocessor is configured to look at those ports. It's all automatic.

As powerful as this is, be aware that you can run into some inadvertent issues that create false negatives. While we can use the built-in service inspectors like http, ssl, ssh, telnet, etc., there is absolutely no indication that we have specified a service header that is valid. This means that if we inadvertently specified "http" rather than "http," Snort will not complain; however, Snort will also never process that rule since there are no inspectors providing data that match that incorrectly typed service name!

Rule Options

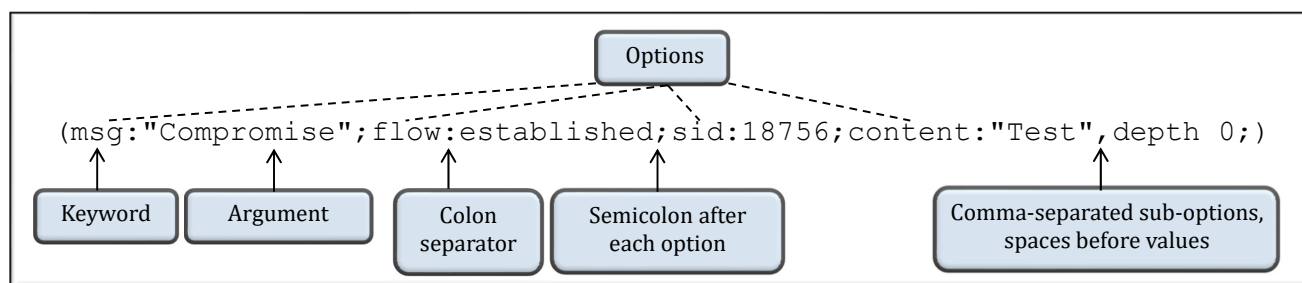
```
(msg:"INDICATOR-COMPROMISE Microsoft cmd.exe banner";  
flow:established, from_client;  
content:"Microsoft Windows", offset 0, depth 17;  
content:"Copyright |28|c|29|", distance:0, within 34;  
content:"Microsoft Corporation", distance 6, within 21;  
classtype:successful-admin; \  
sid:100000001; rev:2;)
```

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

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

Rule options can contain metadata that provides a description and additional information with the alert output, such as references to research the attack and an assessment of the type of attack it is. These metadata options are not used for matching payload, just for output associated with the alert.

Rule Options Syntax



- Rule options made up of one or more options consisting of:
 - Keyword: Reserved value to identify option
 - Colon: Separator between keyword and argument
 - Argument: Value of keyword
 - Semicolon: Separates individual options and denotes end of options list
 - Sub-options are given as a comma-separated list with spaces between the sub-option and value

The rule options section between the left and right parentheses uses reserved *keywords* and associated *arguments*. Just about every keyword requires an argument value; however, there are a few that do not, such as **sameip**, which looks for the same source and destination IPs in the rule header. The keyword and argument are separated by a colon, and individual options are separated by a semicolon. There are some options that take sub-options. An example of this is pictured above with the **content** option. We will explore content options in some depth. Notice that following the content option we have a comma and then the sub-option **depth**. This option has a value of zero, which is specific to following a space.

Each keyword/argument pair comprises an *option*. Make sure that you separate options with a semicolon and denote the end of all rule options with a semicolon too. At one time, omission of the ending semicolon would cause an error, but this is no longer true. Still, it is considered good practice to include it.

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

content Keyword

Examine the packet payload:

- Content can consist of text, hex, or a mix of both.
- Hex data is surrounded by vertical pipes: "|90 90|".
- Can be described using regular expression syntax

Content matching may also be done by exception with an "!" character.

Content modifier sub-options are available.

Multiple content options/values that may have modifiers can be defined.

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

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

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

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

Metadata in Our Rule

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 4444
(msg:"INDICATOR-COMPROMISE Microsoft cmd.exe banner";
flow:established, from_client;
content:"Microsoft Windows", offset 0, depth 17;
content:"Copyright |28|c|29|", distance 0, within 34;
content:"Microsoft Corporation", distance 6, within 21;
classtype:successful-admin;
sid:1000001; rev:2;)
```

Our rule has several metadata arguments and keywords. The **msg** value appears in the alert output as a terse description, including a likeness or hint of the associated filename that stores the rule, "INDICTATOR-COMPROMISE", followed by an explanation of the type of activity discovered. The rule we created is a modified version of one found in the **indicator-compromise.rules** belonging to the directory where Snort and its rules are installed.

There are several more metadata options in the rule, including a classification type that identifies that this traffic is reflective of successful administrator access, a Snort ID of 1000001, and a revision number of 2. A *Snort ID* (**sid**) is required for all rules, and user-defined rules start at 1000001, as lower **sid** values are reserved for rules defined by the Cisco Talos team security experts or by the community.

Writing Rules I

Workbook exercises “Writing Rules I”

This page intentionally left blank.

• An immersion class in speaking packets, protocols, and hex • Ethernet • IPv4 & IPv6 • TCP, UDP, & ICMP • Wireshark & tcpdump fundamentals, filtering, & investigation	Section 2: Signature-Based Detection • Scapy for building tools and simulating signatures & behaviors • Snort & Suricata • Researching common application protocols	• Architecting for massive packet capture & analysis • Overcoming the limitations of signature-based tools • Basic Zeek usage and logs • Threat modelling • Advanced correlation using Zeek	Section 3: Anomaly-Based Detection • Guided scenarios to reinforce all skills • Elaboration on effective deployment, collection, and analysis at large scale • Moving beyond alerts to data, driven analysis, & machine learning	• Challenge questions guide/follow expert analysts' progression through real-world data
---	---	---	--	---

- Packet Crafting for IDS/IPS
- Wireshark Part III
- Application Protocols with Snort and Suricata

Signature-Based Detection

Introduction to Snort

Writing Rules

Content Options

Streams

HTTP and HTTP Rules

DNS and Byte Test

We've successfully written a rule, but our rule is far from optimal. In this section, we will examine some of the more common content options that are used to make Snort run fast!

Optimizing Content Searches

We test with a few packets but think about billions of packets:

- Tuning the rule for efficiency and efficacy

Try to limit how many bytes Snort must examine:

- Designate exact or approximate position where to start content search
- Designate exact or approximate number of bytes where to stop content search
- Capability to specify:
 - The start and/or stop of content from the beginning of the payload
 - The start and/or stop content relative to the end of the previous content match

When creating signature rules in Snort (or any other signature-based IDS, for that matter), it is critical that you keep the big picture in mind. When developing a new rule, we will almost always use a small representative packet capture. This isn't a bad approach, but it can lead to us failing to recognize performance problems! When writing rules and operating an IDS, it is important that you never think of things in terms of a handful of packets; you must always think about billions of packets and how your system will perform in that environment.

It is essential to optimize content searches as precisely as possible. Fundamentally, content searches are almost always the reason that the rule is written. Content values can be modified with keywords that restrict the search to begin at a given position in the packet payload and are contained in a particular number of bytes to restrict the number of bytes of payload searched.

Snort provides for two different reference points when restricting content searches. The first is a reference point from the beginning of the payload. The second reference point is relative to the previous **content** search in a rule. There is a separate set of Snort keywords that identify the beginning and ending locations of content based on whether the search is from the beginning of the payload or relative to the end of the previous content location.

More Efficient Search for Content: Where to Start Content Searches

Microsoft Windows [Version 6.2.7601] **Copyright (c)** 2009 **Microsoft Corporation**
 ↑ **Start:** 0 bytes from beginning ↑ **Start:** 0 bytes from end of previous content ↑ **Start:** 6 bytes from end of previous content (counting starts at 0)

- Search for "Microsoft Windows" at 0 byte offset of payload
- "Copyright (c)" follows, located within next 34 bytes – don't care what other content is in the 34 bytes
- "Microsoft Corporation" follows, located after 6 bytes of some variable content

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

We define the locations and the span of the content we want to include in the rule in this and following slides. Let's examine starting locations first. We want to find the content of "Microsoft Windows" that is at the very beginning of the payload located at the 0-byte offset.

Next, we want to look for the content of "Copyright (c)" beginning at the end of the previous content, "Microsoft Windows". The assumption is that there is some undefined text that may be between the end of the previous content and the beginning of this content. We begin the search immediately after the previous content, defined as relative 0-byte offset.

The final content search is "Microsoft Corporation". This is located at a known number of bytes from the end of the previous payload. This location is identified as relative 6-bytes offset from the end of the previous payload.

Keep in mind that all of these starting positions are offsets. Offsets, like we defined for Berkeley Packet Filter locations for tcpdump, begin the count at offset 0.

More Efficient Search for Content: Where to End Content Search

Microsoft Windows [Version 6.2.7601] Copyright (c) 2009 Microsoft Corporation

↑ 17 bytes ↑↑ Must fall within these 34 bytes ↑ 21 bytes

Now that we have identified the starting location of each content, we have to restrict the number of bytes of content searched. The content of "Microsoft Windows" is exactly 17 bytes. There is some ambiguity about the content and the content length that falls in between "Microsoft Windows" and "Copyright (c)". All we know is that the content of "Copyright (c)" ends within 34 bytes following "Microsoft Windows".

The length of the third content search of "Microsoft Corporation" is 21 bytes precisely. Our rule has a combination of content that has an exact number of bytes and some variance of content where an exact number of bytes cannot be specified, but a maximum number of bytes can be defined.

We begin counting at 1 when supplying the number of bytes of content because there is no such thing as 0 bytes of content length.

Confusion: Value of Starting Position vs. Value of Number of Bytes

content keyword modifiers can specify:

- An exact or approximate starting position of the content value
 - Counting starts at 0.
- An exact or approximate number of bytes of content value
 - Counting starts at 1.
- Counting starts at 0:
 - When you count the offset using tcpdump to specify byte location, start at 0.
- Counting starts at 1:
 - This represents a number of bytes; you cannot have 0 bytes of content.

As noted, there is the potential for confusion of where to start the count for specifying the starting location and the number of bytes of content. The starting position is equivalent to an offset. We have been exposed to offset values before when we defined tcpdump offset filters. Try to remember that as the analogy to avoid confusion.

The expanse of content in terms of number of bytes is a nonzero value. It can be designated as an exact number of bytes in length or a span of bytes in length where the content must be contained. You cannot have a 0 number of bytes of content.

Content Starting Position and Number of Content Bytes

Specify an exact or approximate starting location of content (counting starts at 0):

- **offset**: Relative to the beginning of the payload
- **distance**: Relative to the end of the previous content location

Specify an exact or approximate ending location of content (counting starts at 1):

- **depth**: Relative to the beginning of payload (but Snort is very forgiving here)
- **within**: Relative to the end of the previous content payload

Conceptually equivalent:

- **distance** = **offset**, but default starting location is the last byte of the previous content value
- **within** = **depth**, but default count starting location is after the last byte of the previous content value

Now that we know where to start counting, let's look at the content sub-options you have at your disposal. Snort uses different starting and ending keywords, depending on whether the content location is relative to the beginning of the payload or relative to the end of the previous content payload.

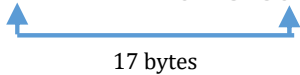
The **offset** sub-option is used when the content starting location is relative to the beginning of the payload. Again, because this is an offset, the counting begins at 0. The **distance** sub-option is conceptually equivalent to **offset**, except it is relative to the end of the previous content location.

The **depth** keyword is used to define a number of bytes of content or content expanse relative to the beginning of the payload. In other words, the content must fall at a byte count before or equal to the value specified. Counting begins at 1 because there is no such thing as 0 bytes. The keyword and value of **depth:17** indicates that the content must be found in the first 17 bytes from the beginning of the payload. Officially, **depth** should refer to the absolute depth into the payload, but Snort is very forgiving here; it will properly treat this as a **within** sub-option.

More Efficient Search from Payload Beginning: Offset and Depth

```
content:"Microsoft Windows", offset 0, depth 17;
```

Microsoft Windows [Version 6.2.7601] Copyright (c) 2009 Microsoft Corporation
01234567891111111
 0123456



depth: Byte count
from beginning
(count starts at 1)

offset: Byte position to
start search from
beginning
(count starts at 0)

Let's look at each of the content values and modifiers in the rule starting with "Microsoft Windows". The content starts at the beginning of the payload at 0 bytes offset; therefore, we specify **offset 0**. There are exactly 17 bytes of content, denoted by **depth 17**.

More Efficient Search End of Previous Content: Distance and Within

```
content:"Copyright |28|c|29|", distance 0, within 34;
```

```
Microsoft Windows [Version 6.2.7601] Copyright (c) 2009 Microsoft Corporation
01234567891111111122222222233333333334444444445
01234567890123456789012345678901234567890
```



distance: Byte position
to start search relative
to end of previous
content
(count starts at 0)

within: Byte count
from end of previous
content
(count starts at 1)

The next content search is "Copyright (c)". The content value is expressed as "Copyright |28|c|29|", which is a combination of text and hexadecimal. Hexadecimal values are delimited by the pipe sign. They represent the left (0x28) and right parentheses (0x29). Are we required to use hexadecimal here to indicate parentheses? No. We could have written this as **content:"Content (c)"**. We have chosen to use the hexadecimal to illustrate how arbitrary bytes can be inserted into content strings.

The location of this content has been described as anywhere within 34 bytes following the content "Microsoft Windows". This means that it can begin anywhere after the end of "Microsoft Windows". Because this is a byte location, the value is 0. The **distance** keyword is used because the content is relative to the previous content location. The content must be within the next 34 bytes, expressed as **within 34**. Again, counting starts at 1 when dealing with the number of bytes.

distance and within

```
content:"Microsoft Corporation", distance 6, within 21;
```

```
Microsoft Windows [Version 6.2.7601] Copyright (c) 2009 Microsoft Corporation
01234567891111111112222222222333333333334444444444555555555566666666667777777777
012345678901234567890123456789012345678901234567890123456789012345678
```



distance: Byte position to start search relative to end of previous content (count starts at 0)

within: Byte count from end of previous content (count starts at 1)

Finally, the content of "Microsoft Corporation" is at a location 6 bytes following "Copyright (c)", as shown below.

```
Copyright (c) Δ2009ΔMicrosoft Corporation
0123456
```



Therefore, **distance 6** specifies the starting location, and **within 21** specifies the exact number of bytes in "Microsoft Corporation". This accomplishes skipping over the 4-digit year and the space before and after the year. A change of year in the banner greeting will not affect the rule.

We've followed the options and concepts of our rule, giving you a good understanding of the most commonly used ones in Snort rules.

Content Modifier `fast_pattern`

Snort gains efficiency using pattern matching algorithm:

- Default pattern matching uses longest specified rule content value to match
- May not always be what you want or most efficient
- By default, looks for the longest static content match first

`fast_pattern` is a **content** keyword modifier that allows other matching options/conditions

Let's look at a couple of Snort options that do not occur in the rule. They can improve performance or use regular expression syntax for content search.

An exhaustive task, in terms of time and necessary processing, is the examination of a reassembled stream payload or a single-packet payload to find the desired content. Snort uses a default pattern matching algorithm to make this process as streamlined as possible. The expected behavior of the pattern matcher is to take the first, longest, non-negative (doesn't contain "!" before the content value) **content** to use as the pattern matching value when multiple content searches are defined in a rule. The **`fast_pattern`** content modifier alters this default behavior to specify a shorter **content** value as the pattern matching value.

The example given in the *Snort User's Manual* is the string "INVITE sip|3A|", frequently found in all *Session Initiation Protocol (SIP)* traffic. Suppose you write a SIP rule that includes that content and another content string that has fewer bytes. It is more efficient for the pattern matcher to search for the other, shorter string first because it is more unique. Less processing is performed when the smaller string is sought, and when found, the packet is then searched for "INVITE sip|3A|."

pcre Keyword

PCRE = Perl Compatible Regular Expressions

Best practice: **Always** couple a PCRE match with a static content match!

```
alert tcp $HOME_NET any -> $EXTERNAL_NET 4444
(msg:"INDICATOR-COMPROMISE Microsoft cmd.exe Banner pcre rule test";
flow:established, from_client; content:"Microsoft Windows", offset 0, depth 17;
content: "[Version]"; pcre:"/[Version \d\.\d\.\d\d\d\d/";
sid:100000009;)
```

```
sans@sec503: snort -A console -K none -q -r inbetween.pcap -c pcre.rule
06/16-09:21:38.272481 [*] [1:100000009:0] INDICATOR-COMPROMISE Microsoft
cmd.exe Banner pcre rule test [*] [Priority: 0] {TCP} 192.168.11.17:1770 ->
192.168.1.8:4444
```

SANS

pcre.rule
inbetween.pcap

SEC503: Intrusion Detection In-Depth 88

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

For instance, in the above rule, we would look for a **content** of "[Version" as an anchor for the regular expression of "[Version #.#.####". The regular expression in the rule looks for "[Version ", followed by a pattern of a single digit, following by a period, followed by a single digit, followed by a period, followed by four digits.

This anchor content is the fixed part of the string that does not change. This allows the pattern matcher to efficiently find the content before the **pcre** content containing that same string is parsed.

The regular expression syntax can be somewhat baffling at first if you have never been exposed to it. The use of **pcre** in a rule can be very effective for finding variable content anchored by known content. It would be beneficial to learn about regular expressions if you anticipate writing or understanding rules. Regular expressions are quite complex, and a thorough discussion of them is out of the scope of this class. For tutorials on regular expressions, check the book *Mastering Regular Expressions*, published by O'Reilly, as well as the following website: <http://www.php.net/pcre>.

- An immersion class in speaking packets, protocols, and hex - Ethernet - IPv4 & IPv6 - TCP, UDP, & ICMP - Wireshark & tcpdump fundamentals, filtering, & investigation	Section 2: Signature-Based Detection - Scapy for building tools and simulating signatures & behaviors - Snort & Suricata - Researching common application protocols	- Architecting for massive packet capture & analysis - Overcoming the limitations of signature-based tools - Basic Zeek usage and logs - Threat modelling - Advanced correlation using Zeek	- Guided scenarios to reinforce all skills - Elaboration on effective deployment, collection, and analysis at large scale - Moving beyond alerts to data, driven analysis, & machine learning	Challenge questions guide/follow expert analysts' progression through real-world data
---	--	---	---	---

- Packet Crafting for IDS/IPS
- Wireshark Part III
- Application Protocols with Snort and Suricata

Signature-Based Detection

Introduction to Snort

Writing Rules

Content Options

Streams

HTTP and HTTP Rules

DNS and Byte Test

All the rules we have written in our labs so far are limited; they will only find the content match if the entire content string appears in a single packet. What if we need to look at the stream as a whole? Let's talk about the **frag3** and **stream5** preprocessors.

flow Keyword

Detection can be limited by the direction specified in a rule:

- We can either specify a second rule in the opposite direction...
- Or we can use the `<>` direction operator rather than `->`
- Alternatively, something like `alert http ()` is automatically bidirectional.

Most common state option:

- `flow:established` – Stateful inspection has marked session as established.
 - No alert will be generated unless an established connection has been detected.
- We can specify a direction:
 - `flow: established, to_server`, for example

Let's look at the **flow** option in more detail because it is important and often used. Rules can be written to take advantage of the stateful inspection reassembly capabilities available in Snort. The **flow** option requires that the TCP and/or UDP reassembly is enabled (which is the default). There are some general types of options that are available for the **flow** keyword; we cover only state options and direction options.

The state options apply to established or unestablished/stateless TCP or UDP sessions. "Established" means after the three-way handshake for TCP or as a series of packets using the same source and destination IPs and ports for UDP. These are sessions that have been reassembled by Snort. If we miss part of the initial handshake, Snort may incorrectly decide that a connection isn't really established.

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

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

When you are using the **to_server** and related options, it is quite likely that you should use the `<>` direction operator rather than the `->` operator in the rule header. The `<>` applies the options bidirectionally. Otherwise, your rule might miss content that you would intuitively expect it to alert on!

In Snort 3, if we are using a service header, we need not be concerned about the direction operator. Indeed, as you know, we never specify one. These rules are inherently bidirectional.

Interpreting an Alert

How do you know which rule caused an alert to be logged?

What are those numbers about?

```
06/16-09:21:38.272481 [**] [1:100000009:0] INDICATOR-COMPROMISE Microsoft
cmd.exe Banner pcre rule test [**] [Priority: 0] {TCP} 192.168.11.17:1770 ->
192.168.1.8:4444
```

Generator ID	SID of the rule	Version # from the rule
1: Signature		
2: Tagging system		
105: BackOrifice		
106: RPC Decode		
111: Stream4		
112: Arpspoof		
116: Internal decoder		
...		

Before we continue, we thought it would be important to discuss the format of alerts, just in case your instructor hasn't talked about it yet. While the majority of the alert is pretty intuitive, the square brackets with three colon-separated values can look very mysterious.

The first number in that field represents the part of Snort that generated the alert. The most common thing to find here is "1," which means that it was the result of a signature match, but other values are possible too. For example, a "2" would indicate that the **tagging subsystem** caused this packet to be logged. This means that this packet did not actually match a rule, but the tagging system caused it to be logged. Why would this happen? This would occur if a rule, for example, included a **tag** option, instructing it to log the next 100 packets after an alert was logged (**tag: session, 100, packets, exclusive**).

The second number is the Signature ID, or Rule ID, if you prefer. Numbers less than 1,000,000 are reserved for use by the Snort team and rule feeds, while numbers above this are for your use for local rules.

The last number is populated from the **version** metadata option, which we would strongly encourage you to use when writing Snort rules!

My New Rule Doesn't Work! What Now?

Give up, panic, or **debug**.

Three possible reasons for failure:

- The rule is wrong.
- The **snort.lua** file is not properly configured.
- The traffic/pcap you are capturing or reading has an issue.

A key to avoiding issues is to make just one change at a time.

- Every time you make a change, update the revision and test the rule.

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

However, don't discount some other explanations, such as a configuration file that does not handle the type of traffic you are using or even a pcap that is corrupted in some way. You might not have the required preprocessor loaded in your **snort.lua**. For example, if the reassembly options are incorrect or if your **\$HOME_NET** and **\$EXTERNAL_NET** variables are not properly set, you will receive no alert. A final check is to make sure that the file location of the new rule is included in the Snort configuration file.

As far as a corrupted pcap, make sure there are no checksum errors. By default, Snort drops packets with an incorrect checksum. After all, that is what the destination host does. Snort can be easily evaded if it does not behave in the same manner as the receiving host.

Rule debugging is a methodical iterative process. First, delete all the rule options except for the **msg** and **sid**, leaving only the header part of the rule. It's possible that something is wrong in the header and is easily corrected. Run Snort again using the new rule. If the rule fires with the rule header alone, add the rule options back one by one, starting with the first option listed in the rule, rerunning Snort after each change until the alert no longer triggers. Our advice would be to use this practice when writing rules rather than waiting for something to break. This approach can save you a great deal of time.

Unavoidable: False Negatives and Positives

<u>False Negative</u>	<u>False Positive</u>
• No detection of attack/event • Possible reasons ◦ Packet loss ◦ No rule written to handle the attack ◦ Badly written rule ◦ Attack obfuscation ◦ Overloaded configuration ◦ Critical path failure	• Rules go off for nonevents • Possible reasons: ◦ Loosely written rules (too broad) ◦ Unnecessary rules



False negatives and *false positives* are a fact of life. False negatives happen when the IDS/IPS misses an event that should have been detected. There are several general causes of false negatives. One of them is packet loss. Packet loss can be related to hardware, software, or a combination of both. Sensor hardware issues can be caused by network interface cards that cannot keep up with the traffic, an overloaded hardware bus that transports the data internally, insufficient memory and/or processor(s) speed, overloaded capture hardware such as an oversubscribed switch, or a combination of one or more of these.

Rules or the lack thereof can be the source of false negatives. You cannot detect a zero-day for which there is no rule! Poorly written rules also lead to both false negatives and false positives. Obfuscation can cause a false negative if the IDS/IPS does not properly interpret or decode the payload.

The configuration of Snort, such as the number and types of preprocessors selected and values for those preprocessors, may cause Snort to slow down and potentially drop packets. For instance, by default, the **stream5** preprocessor looks for traffic over a subset of the possible 65,535 ports. You could alter this to make Snort look at all ports to lower your risk of a false negative, but you run the risk of slowing Snort down, possibly to the point where a false negative occurs because Snort is not able to keep up. It is a balancing act.

The critical path is the portion of a piece of code that executes the most frequently in an application. Within Snort, this path is decoding, detecting, and reporting on the traffic. If this critical path is too slow or is accessed too frequently, packets may be lost because the sensor can't keep up. As much as possible, restrict the sensor to sniffing and analyzing. It should not be concerned with the post-processing of alerts.

A false positive occurs when a meaningless event triggers a rule. A false positive alert can occur if a rule is too broad, causing it to erroneously fire on something that is not noteworthy. Another reason is that the rule may be well written, yet not apply to the given site or environment. For instance, suppose there is a rule that pertains to IIS, but you run Apache. An IIS alert that is triggered is basically meaningless to the analyst because it does not represent a threat in the environment.

False Negative Solutions

Packet loss:

- Faster, more powerful hardware
- Refine the critical path to minimize processing not connected to sniffing/analyzing
- Remove bloated preprocessors, tune the settings on relevant ones
- Remove rules with no applicable network traffic

Analysis failures:

- Place sensors in appropriate locations
- Write rules for a given vulnerability
- Include all relevant rules
- Enable all necessary preprocessors
- Update rules frequently
- Suppress repeated identical alerts to avoid analyst overload

False negatives have several causes. The first category is related to packet loss issues. Packet loss can occur because of slow or underpowered hardware, anywhere from traffic capture to post-analysis of writing to disk. How can this type of packet loss be avoided? Generally speaking, the answer is to spend either a lot of money or a lot of thought. Putting Snort on faster hardware with more memory will have positive effects on performance.

Other packet loss issues are a result of overburdening Snort where packets get bottlenecked and dropped. These are issues such as a critical path overload (for instance, using several output *plugins*); pare those down so that Snort is predominantly sniffing and analyzing. Plugins are modular pieces of code written for Snort that allow programmers to extend the functionality of the program. There are three types: preprocessor, detection, and output.

As we've talked about, delete bloated unnecessary preprocessors that have no relevant traffic. Another preprocessor-related problem is using relevant preprocessors but tuning the settings poorly—say, for instance, by increasing the maximum concurrent sessions that **stream5** can process without increasing the memory allocated for TCP processing. It is important to consider the implications of changing a setting because it may have unexpected consequences.

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

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

The point is that Snort is not self-aware; it is the job of the analyst to discover detection issues and correct them.

Dealing with Alert Overload for Same Activity

Per individual rule:

- Rule detection filters: Threshold count and/or time for generating alert for a specific source or destination IP

Defined in threshold.conf for one or more rules:

- Rate Filtering
 - Specifies a new rule action after given rate exceeded
- Event Filtering
 - Specifies the number of times a particular event is logged per number of events in a time interval
- Event Suppression
 - Specifies rule(s) that will not fire based on Snort ID and/or IP address

It is possible that an especially noisy alert can be a distraction that may prevent you from seeing more meaningful alerts, leading to a false negative situation. There are several different ways to limit or suppress alerts when they become noisy. We will discuss the *rule detection filter* method only. If you are interested in the others, the Snort Manual explains them in great detail, giving examples of each.

Writing Good Rules

Use longest content match, or most unique shorter content match via **fast_pattern**

Avoid rules with no content match, negative content match, or pcre with no content match anchor.

Place "bail" conditions in the rule; limit the number of bytes tested.

`flow, offset, depth`

To sum up the previous slide, whenever possible, supply an appropriate destination port because this is the earliest efficiency that can be realized. The **any** destination port requires every packet with a given protocol to be examined.

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

Use options in the rule that can offer an early exit to processing, such as **flow**. You see how rules that have no destination port, or no content or negative content, create a generic "bucket" of rules causing all packets that have arrived at that bucket path to be analyzed. It's analogous to the programming statement like a "switch" or "case" that lists all possible values for a given variable and typically has a catchall default statement in the event that none of the values matches. The packet may eventually end up in the catchall category, requiring it to go through wasteful checks.

Writing Rules II

Workbook Exercises "Writing Rules II"

This page intentionally left blank.

• An immersion class in speaking packets, protocols, and hex • Ethernet • IPv4 & IPv6 • TCP, UDP, & ICMP • Wireshark & tcpdump fundamentals, filtering, & investigation	Section 2: Signature-Based Detection • Scapy for building tools and simulating signatures & behaviors • Snort & Suricata • Researching common application protocols	• Architecting for massive packet capture & analysis • Overcoming the limitations of signature-based tools • Basic Zeek usage and logs • Threat modelling • Advanced correlation using Zeek	• Guided scenarios to reinforce all skills • Elaboration on effective deployment, collection, and analysis at large scale • Moving beyond alerts to data, driven analysis, & machine learning	• Challenge questions guide/follow expert analysts' progression through real-world data
---	--	---	---	---

- Packet Crafting for IDS/IPS
- Wireshark Part III
- Application Protocols with Snort and Suricata

Signature-Based Detection

Introduction to Snort

Writing Rules

Content Options

Streams

HTTP and HTTP Rules

DNS and Byte Test

Since Snort and all other signature-based IDS/IPS solutions are primarily optimized for content matching, the **content** rule option is the most frequently used option you will find. Close behind this are the HTTP-related content options. While more and more of our HTTP traffic is being run over TLS today, we will deal with TLS as a separate issue in Book 4. In this section, we will seek to understand the HTTP protocol in all its common variants and then explore detection options in the lab exercises.

Detection Methods for Application Protocols

Protocol decode:

- Detect that a given protocol is in use and parse/examine it as the application does.

Pattern matching:

- Look for one or more strings or regular expressions in packet payload.

Anomalous behavior:

- Examine connectivity patterns possibly for a specific volume/threshold of packets.

There are several different ways to examine or detect malicious or anomalous behavior in network traffic. The three methods listed on the slide (protocol decode, pattern matching, and anomalous behavior) may be found in many IDS/IPS products, and they may be used in some combination or other to find a single malicious attack.

Protocol decode is the most labor intensive in terms of development effort for the vendor or developer. Good protocol decode involves identifying a given protocol (say, HTTP), not just on the basis of the port over which it is transported. There must be some identifying characteristic of the payload that uniquely defines it as a given protocol. Using the HTTP example, there would be a GET or POST or some other HTTP method request. The IDS/IPS must then parse and analyze the protocol as the receiving application would. Although this is the most difficult type of detection to develop, it probably is the most accurate in terms of specifying a signature or rule for the attack. Say that the IDS/IPS can isolate the field that specifies the URL request and that an attack involves an overly long URL. This would be a rather trivial signature to write. By the way, Wireshark/tshark is based on dissectors that decode protocols.

Pattern matching allows the specification of a string or regular expression to search for in packet payloads, often with either a precise or relative position to search in the packet. This is more generic than the protocol decode method; however, there are times when there is an unconventional protocol or file format that needs to be examined. Pattern matching may be prone to false positives, depending on the use and signature.

Finally, there is activity that cannot be specified by protocol decode or pattern matching alone. *Anomalous behavior* typically examines network connectivity patterns in terms of volume or threshold of packets, protocols, and more perhaps in a given time. Exfiltration, tunnels, and command and control falls into this category. These attacks involve larger than normal connectivity patterns of traffic within a specified time period. One big issue associated with network behavior analysis is determining what baseline "normal" traffic is in order to deem outlier traffic anomalous.

Standard HTTP Request/Message

HTTP Verb (GET, POST, PUT, etc.)

HTTP Version

Optional
Headers

```
GET / HTTP/1.1
Host: www.google.com
User-Agent: Mozilla/5.0 (X11; Linux i686; rv:15.0) Gecko/20100101 Firefox/15.0.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Connection: keep-alive
Cookie:
NID=64=bLCb1rCTzwCwCZJMHvV2dFiKcvy0phB7mg7uUnh574HQs4qEdDqsLI2M0H0EgARbH5T9hwh0Edz9CuA5(
RWeomf8IT4J-u7g_7ZKIwJSkP13-DKZfpQ9LJwF28xUx9LBAFatF_dIHV8vHE7mYseM-BqoXwYs1BYCZmmnNPS3:
SID=DQAAAJ0AADNCTMSN_55sL1pmOe72FJ7Q21dtV5nfikvAv7N84ZztLomd6j3nD3wIdWdDXRkwZrTk6Wxr7V;
nblgVnToLW-vqsSEYMj17IxaZib8qp3pEct3CMPtsuMjHqemTKhPI4iPzTa3a5awrQRR327LIVVRC1EwWgFGWm\
UXg3n0tuYszP12WqZZHpN-8RkIwwQS; HSID=A_x5Xx6y2ipriCBth; APISID=6T91e1j1-GXj2qcs/AjX1eC3(
PREF=ID=1c6b4a77323d4e23:U=3fde0d31968682c7:FF=0:LD=en:TM=1348857405:LM=1348857748:GM=1
```

Host name - Required in HTTP 1.1 for virtual hosting

Blank line (Possibly followed by a message body up to the Content-length)

SANS



SEC503: Intrusion Detection In-Depth 100

Hypertext Transfer Protocol/Hypertext Transfer Protocol Secure (HTTP/HTTPS) is the most prevalent protocol used. It provides a broad range of functionality, and it continues to get more and more complex. Initially, attacks on HTTP/HTTPS were aimed at the service itself. Today, attacks tend to target the browser as a means to compromise the endpoint.

HTTP has a rather simple protocol format. The message body of an HTTP request or reply is where things get wild. Server applications and browsers (usually via JavaScript) are plagued with vulnerabilities at the application layer, attracting all kinds of attacks. We'll look at just a few of these issues because this discussion of HTTP is concerned more about the format of the protocol and less with the many attack types.

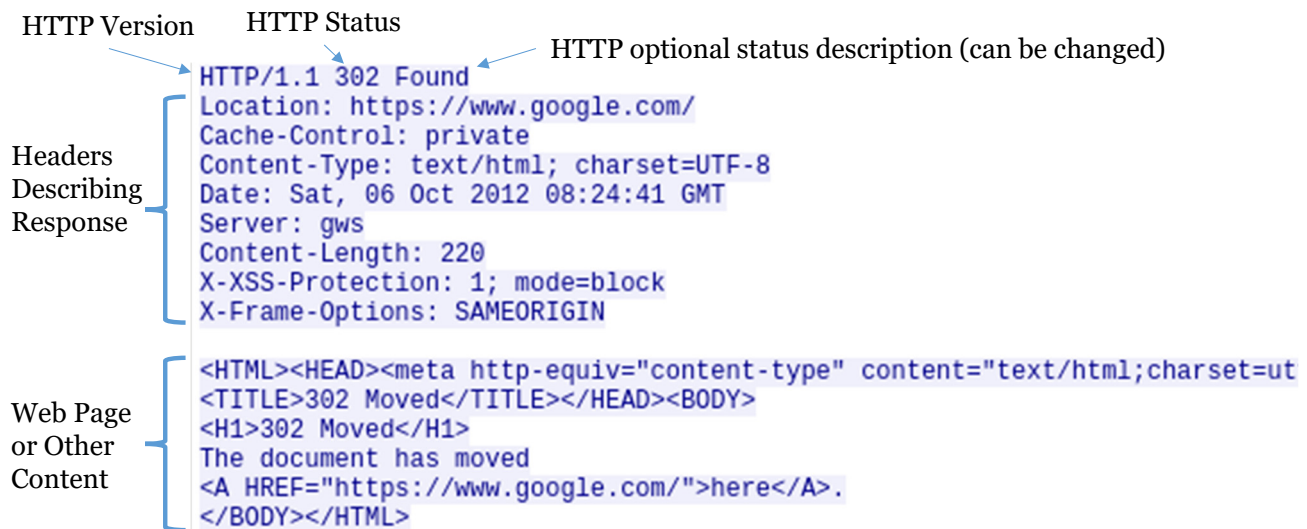
HTTP performs simple exchanges of HTTP requests followed by HTTP responses. HTTP is defined as a stateless protocol. Any stateful activities within an application must be simulated or maintained by the application itself, usually through the use of a session token.

An HTTP request begins with a start line that includes a *method* or *verb*, a *Universal Resource Locator (URL)*, and the HTTP version supported by the client. If HTTP/1.1 is being used, a *Host* header is required, which is used to identify which virtual web service the client is requesting. Following these, any number of optional headers may appear, including things like HTTP cookies, information about encoding and formats that are supported, etc. Following the headers, a single blank line will appear, which is how the server knows that it has reached the end of the request.

There are several different HTTP methods, though most of the time **GET** or **POST** is used. A **GET** method requests some kind of resource or document identified by the URL from the server. **POST** also sends a request and also sends data to the specified URL in the body of the request, below the headers. Data can be sent in the **GET** as well but is limited to the headers and the parameters in the URL itself.

HTTP is a *line-based* protocol. This means that the protocol uses a new line character as a delimiter between different elements, such as the HTTP start line and header lines, and between those and the message body.

Standard HTTP Response/Message



The HTTP response is similarly simple. The header begins with a single line that describes the version of HTTP in use, followed by a status code, and finally a text description of the status. We'll look at status codes on the next page. The status description is completely arbitrary and can be changed to any arbitrary string. It turns out that the clients really only care what the number is, not the text description.

Following this first line, we find a variety of headers. While these are fairly standard, they can mostly be viewed as optional. One of the more important and interesting headers for us would be the **Content-Type** header, which defines the type of data being sent back. This value is what will trigger a download rather than rendering a web page, for example.

After the headers, we again have a single blank line, after which we have the body of the response. In the example in the slide, this contains a basic web page that is providing redirect information. If the request had been for a JavaScript file, we would expect to see JavaScript, CSS if it were for CSS, a binary image file if it had been for an image, and so on.

One last item of note is that most web servers these days will compress the data portion if an image or something similar is being sent. This is great for efficiency, reducing the number of packets sent. This is less ideal from an IDS/IPS perspective since it means that the IDS must be able to decompress the data on the fly in order to perform signature matching; otherwise, there is no way to find strings in the data.

HTTP Status Codes

Web technology for developers > HTTP > HTTP response status codes

 Change language

Table of contents

Information responses
Successful responses
Redirection messages
Client error responses
Server error responses
Browser compatibility
See also

Related Topics

HTTP

Guides:

- ▶ Resources and URIs
- ▶ HTTP guide
- ▶ HTTP security

HTTP access control (CORS)

HTTP authentication

HTTP caching

HTTP response status codes

HTTP response status codes indicate whether a specific [HTTP](#) request has been successfully completed.

Responses are grouped in five classes:

1. [Informational responses](#) (100 – 199)
2. [Successful responses](#) (200 – 299)
3. [Redirects](#) (300 – 399)
4. [Client errors](#) (400 – 499)
5. [Server errors](#) (500 – 599)

The below status codes are defined by [section 10 of RFC 2616](#) . You can find an updated specification in [RFC 7231](#) .

If you receive a response that is not in this list, it is a non-standard response, possibly custom to the server's software.

Information responses

[100 Continue](#)

This interim response indicates that everything so far is OK and that the client should continue the request, or ignore the response if the request is already finished.

129

There are many different references that you can find that describe the various HTTP status codes. While RFC-7231 is the official definition, we know that RFCs are not always the easiest to read. One of the clearest references is available from Mozilla at <https://developer.mozilla.org/en-US/docs/Web/HTTP/Status>.

There is no need to memorize these codes. Instead, knowing the general categories, which are defined by the range of numbers that a result code appears in, is sufficient. The categories are:

- 100–199 – Informational responses
- 200–299 – Success
- 300–399 – Some kind of redirect or additional action required
- 400–499 – Client errors (which include things like client requesting a page that is missing, or requesting a page that requires authorization without presenting a credential, or even attempting to access a page that is forbidden)
- 500–599 – Server errors (which include things like the web application crashing, a gateway being unavailable, a proxy failing, etc.)

HTTP Detection Challenges

- HTTP is a scary nightmare for IDS/IPS products:
 - It encompasses a broad range of features.
 - It is the most prevalent protocol on most networks.
 - And it's used to carry many other higher-level application protocols!
 - The session can be encrypted.
 - It may become a delivery mechanism for attacks on the browser/client or server.
 - Wide variety of types of browsers/servers, each with their own issues
 - Users demand it.

If you think that your IDS/IPS solution is going to provide a robust defense for all kinds of HTTP malice, think again. So many functions and features are associated with HTTP/HTTPS that it is practically indefensible. In many networks, the majority of the packets crossing the network are associated with HTTP/HTTPS. Inspection of these packets for all the many signs of attack is nearly impossible, especially in high-bandwidth networks and inline IPS solutions.

Encrypted sessions cannot be easily examined without taking significant actions. To make things worse, the attack or vulnerability may not be in the HTTP or HTTPS protocol or server, but in the application above that since HTTP is frequently used to carry other arbitrary protocols, including video conferences, voice chat, and more. In essence, HTTP/HTTPS may be considered the "mule" for many different applications and, as a result, attacks.

Malicious executables can be carried over HTTP/HTTPS back to the client and browsers. Many browsers have extensions that allow other software such as Flash to be installed, possibly making them more vulnerable to attack. SQL injection or cross-site scripting attacks are often possible using HTTP/HTTPS to carry the malicious payload from the server. Also, users think it is their constitutional right to have HTTP/HTTPS available, and only the most draconian sites forbid it.

Further, there is a proliferation in the numbers and types of browser and web server solutions. Each has its own implementation along with vulnerabilities and issues. Web application firewalls may better help defend web servers against attacks or to assist in detecting what the IDS/IPS misses. Host-based antivirus may help with client defense, though it may miss many client-side attacks, too.

HTTP/2

Successor to HTTP/1.0 and HTTP/1.1

RFC 7540 describes the purpose of HTTP/2 as "optimizes the transport for HTTP semantics"

Based on Google protocol SPDY

Improves efficiency of processing with:

- Multiplexing of concurrent activity in mechanism called streams
- Compression of HTTP headers to reduce packet payload
- Server push in anticipation of client need
- TLS "hello" exchange uses Application-Layer Protocol Negotiation (ALPN)

Originally, HTTP was a simple text-based protocol that did not require efficient synchronization with the TCP transport layer. HTTP has become the most common application protocol, requiring more bandwidth to carry more sophisticated payloads. The catalyst for the development of HTTP/2 was because HTTP/1.x no longer rides seamlessly over TCP. HTTP/2 is an enhancement for efficiency's sake.

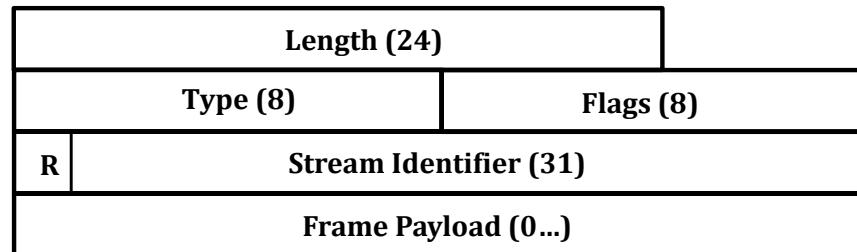
HTTP/1.x is inefficient because multiple concurrent sessions are not possible. HTTP/1.0 processes a single outstanding request for some HTTP-related task in an individual TCP session. HTTP/1.1 uses a mechanism known as *pipelining* to piggyback outstanding requests in a single TCP session. However, both require multiple TCP sessions per HTTP transaction, creating additional TCP overhead of establishing and closing new TCP sessions. RFC 7540 Hypertext Transfer Protocol Version 2 (HTTP/2) summarizes the efficiencies, stating that HTTP/2 "optimizes the transport for HTTP semantics." The main optimization is that it uses a single TCP session for an entire HTTP exchange.

HTTP/2 is based on a Google-created protocol known as *SPDY*, which can be considered a precursor to HTTP/2. Many of the efficiency mechanisms included in HTTP/2 were pioneered in SPDY. SPDY has some differences with HTTP/2 because SPDY makes encryption the standard for network communication and performs header compression using algorithms other than either gzip or deflate.

HTTP/2 interleaves processing tasks using a concept known as a *stream*, permitting both the client and server to communicate concurrently. There is flow control for resources so that a request is sent only when it can be processed. Streams can be prioritized to allow more important streams to be processed first. Dependent streams can be suspended until all antecedent streams complete.

HTTP headers are compressed in HTTP/2 with the HPACK process. Another means for achieving efficiency is known as a server push, where the server preemptively sends responses to the client based on anticipated processing. A final efficiency is used by a browser that supports HTTP/2 encrypted traffic, employing *Application-Layer Protocol Analysis (ALPN)*, an add-on to the TLS protocol. ALPN combines the TLS **hello** protocol negotiation process so that single, rather than two different client/server, round-trip exchanges are necessary.

HTTP/2 Frame Format



HTTP/2 protocol encapsulation unit called a frame

HTTP/2 packets are termed "frames." In fact, the existence of a frame can be considered a new protocol layer between TCP and HTTP. The underlying fundamentals, such as HTTP request methods, headers, data, and other HTTP semantics, have not changed; only the way they are communicated across the network has.

HTTP/2 runs on port 80 for cleartext and 443 for encrypted traffic. Each frame has a 9-byte frame header that has a variable-length payload. The length field indicates the length, in bytes, of the payload, not including the header. The type indicates the frame type; there is a different frame composition based on the assigned type.

Many of the different frame types behave much like a TCP session. For instance, the *HEADER* frame type 0x01, like a TCP SYN, signals that a new stream is to be opened. It carries an HTTP header payload. The *GOAWAY* frame type 0x07 value, like a TCP FIN, signals the last frame is being sent. Like TCP, the other endpoint must send a GOAWAY too, indicating that it will send no more data.

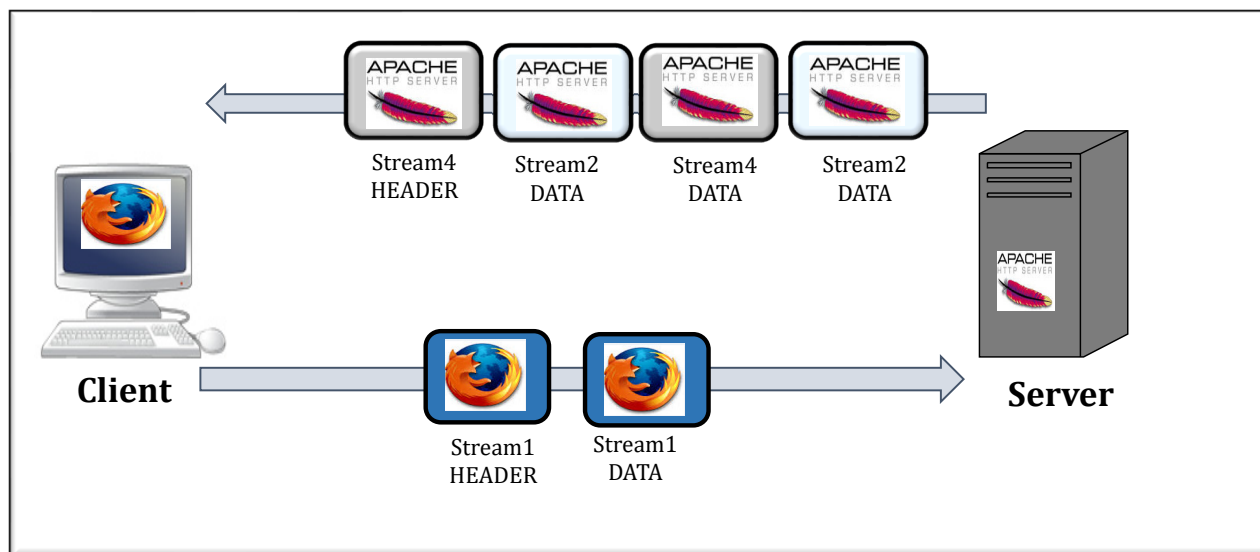
The *RST_STREAM* frame type 0x03, like the TCP RST, is used to immediately abort a stream. A *PING* frame type 0x06, like the TCP keepalive, can determine whether the other endpoint is still alive, measure round-trip time, and refresh the connection through the firewall. A *WINDOW_UPDATE* frame type 0x08, like the TCP window, is used for flow control. The *CONTINUATION* frame type 0x09 indicates a fragmented header frame.

Of the other stream types, one of the most common is *DATA*, type 0x00. This indicates what you might think of as the "Body" of the request. A *SETTINGS* frame type 0x04 assigns different configurations such as maximum concurrent sessions and maximum frame size.

There is a 1-bit reserved field with no assigned use currently. This value must be 0x00 when sent, and the value must be disregarded when received. A unique stream identifier value is a 31-bit unsigned integer assigned to a single frame or related frames. Client-initiated streams must begin with an odd number, and server-initiated streams must begin with an even number.

Finally, padding is optional and may be used to protect against some specific attacks by obfuscating the length of the frame content.

Multiple HTTP/2 Streams per TCP Connection



The diagram provides a pictorial representation of what transpires in a single TCP session composed of multiple individual streams. It depicts the frame layering where the streams consist of either headers or data. It appears that this is the middle of an HTTP/2 connection. You see streams with even value identification numbers 2 and 4 that were initiated by the server and an odd value identification number stream 1 initiated by the client. We may speculate that frames associated with missing stream 3 may have either a higher or lower priority than 1, 2, and 4 and have already been processed or are pending processing.

There are concurrent streams in the single TCP connection that must be reassembled, using the stream identification number, by the receiver. The stream ID can be compared to a TCP sequence number that acts to reassemble received, and possibly out-of-order, stream frames. There is also a parallel with IP fragmentation, where HTTP/2 uses the CONTINUATION type on multipart header frames.

This approach turns out to be far more efficient. Do not read that to mean, "faster." This is no faster than HTTP 1.1. The efficiency is on the server side. Rather than having to support many simultaneous and consecutive connections from each client, with HTTP/2, each client establishes one and only one connection to the server. All the resources that would have been requested over separate connections are now requested as different *streams* over this single connection (as described above). This means that the server needs to maintain only a single socket per client, which is much more efficient and allows a single server to handle more clients simultaneously.

HTTP/2 GET Request

No.	Time	Source	Destination	Protocol	Length	Info
4	2015-02-18 10:35:15.244677	192.168.0.192	106.186.112.116	HTTP	233	GET / HTTP/1...
5	2015-02-18 10:35:15.527889	106.186.112...	192.168.0.192	TCP	66	80→42895 [AC...
6	2015-02-18 10:35:15.528227	106.186.112...	192.168.0.192	HTTP	140	HTTP/1.1 101...
7	2015-02-18 10:35:15.528235	192.168.0.192	106.186.112.116	TCP	66	42895→80 [ΔC...

Transmission Control Protocol, Src Port: 42895, Dst Port: 80, Seq: 1, Ack: 1, Len: 167
Hypertext Transfer Protocol GET / HTTP/1.1\r\n User-Agent: curl/7.41.0-DEV\r\n Host: nghttp2.org\r\n Accept: */*\r\n Connection: Upgrade, HTTP2-Settings\r\n Upgrade: h2c-14\r\n HTTP2-Settings: AAMAAABKAAQAAP__\r\n \r\n [Full request URI: http://nghttp2.org/] [HTTP request 1/1] [Response in frame: 6]

```
GET / HTTP/1.1
User-Agent: curl/7.41.0-DEV
Host: nghttp2.org
Accept: */*
Connection: Upgrade, HTTP2-Settings
Upgrade: h2c-14
HTTP2-Settings: AAMAAABKAAQAAP__
```

```
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Upgrade: h2c-14
```

```
.....d.....PRI * HTTP/2.0
```

```
SM
```

```
.....Z..
```

SANS



http2-upgrade.pcap

SEC503: Intrusion Detection In-Depth 107

Let's look at the way an HTTP/2-capable client communicates to the server that it wants to upgrade to HTTP/2. On the left of the slide, you see Wireshark's interpretation of the packet containing the GET request. Before we discuss the GET request, look on the right of the slide to see the reassembled HTTP/2 session. This is an example of an HTTP/2-enabled client starting a web session with a server whose HTTP version is unknown to the client. We'll look at the situation in the next slide in which a client has prior knowledge that a web server supports HTTP/2.

The GET request contains version HTTP/1.1 because it does not know if the server supports HTTP/2. There is an *Upgrade* header that indicates the desire to use HTTP/2. A designation of "h2" means that the session will be over TLS. A designation of "h2c" would be necessary if it were in cleartext. The "14" that follows is an indication that the software used is based on the interim draft 14 version of the HTTP/2 RFC.

The server supports HTTP/2 and indicates via a return code of 101 that it is switching protocols to HTTP/2. A server that does not support HTTP/2 simply ignores the upgrade request and responds with "HTTP/1.x 200 OK."



HTTP/2 is only supported in Wireshark 2 and above.

wireshark http2-upgrade.pcap

HTTP/2 GET Request Prior Knowledge

```

▼ HyperText Transfer Protocol 2
  ▼ Stream: Magic
    Magic: PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n
  ▼ Stream: SETTINGS, Stream ID: 0, Length 12
    Length: 12
    Type: SETTINGS (4)
    ▶ Flags: 0x00
    0... .. = Reserved: 6
    .000 0000 0000 0000 0000 0000 0000 0000 = Stream Identifier
    ▶ Settings - Max concurrent streams : 100
    ▶ Settings - Initial Windows size : 65535
  ▼ Stream: HEADERS, Stream ID: 1, Length 57
    Length: 57
    Type: HEADERS (1)
    ▼ Flags: 0x05
      ... ..1 = End Stream: True
      ... .1.. = End Headers: True
      ... 0... = Padded: False
      ..0. .... = Priority: False
      00.0 ..0. = Unused: 0x00
    0... .. = Reserved: 6
    .000 0000 0000 0000 0000 0000 0000 0001 = Stream Identifier
    [Pad Length: 0]
    Header Block Fragment: 418aa0e41d139d09b8c8000f820494
    [Header Length: 197]
    ▶ Header: :authority: localhost:3000
    ▼ Header: :method: GET
    0070 0c 04 00 00 00 00 00 00 03 00 00 00 64 00 04 00
    0080 00 ff ff 00 00 39 01 05 00 00 00 01 41 8a a0 e4
    0090 1d 13 9d 09 b8 c8 00 0f 82 00 02 45 54 0a 47
  
```

Header compression

SANS



http2-priorknow.pcap

SEC503: Intrusion Detection In-Depth 108

A client that has prior knowledge that a server supports HTTP/2 does not send the HTTP/1.1 header to signify an upgrade to HTTP/2. The client sends a connection preface of "PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n" to inform the server that it intends to begin an HTTP/2 session. Although not shown in the packets pane in this slide, SETTINGS, HEADERS, and DATA types of frames are then sent in this particular exchange.

One thing to note is that HEADER-type "Header" values are somewhat different from HTTP/1.x headers. There are header fields known as pseudo-headers, such as the one to denote a GET method. If you recall, HTTP/1.x had a start line that designated the method and target URL. An HTTP/2 header field that begins with a colon is a *pseudo-header*. A pseudo-header is not an actual HTTP header field; it's used to convey metadata such as the ":method" for the HTTP method and ":path" to express the URL.

You can see that the headers are not text-based; they are in compressed format. For instance, selecting the pseudo-header ":method" with a value of GET in the Wireshark packets pane translates to 0x82 in the bytes pane. You would see the ASCII representation of 0x47 45 54 in HTTP/1.x.



The slide output was produced using Wireshark as follows:

wireshark http2-priorknow.pcap

HTTP/2 Detection Challenges

New protocol requires IDS/IPS/tools that are HTTP/2-aware and can decode it

- Our tools this week (Wireshark, Snort 3, Zeek) are HTTP/2 aware.
- Not all tools are!

Current browser versions are encrypting the HTTP/2 session by default.

- This represents a larger challenge.
- If we can't read the data, it doesn't matter if our tool supports the protocol.
- More on this in Book 4

As if there were not enough challenges to HTTP interpretation already, we now must consider HTTP/2 as an evolving protocol. It is different enough from HTTP/1 to require new decoding methods due to the frame protocol, header compression, possible header fragmentation, binary data in the HTTP body, a different format for indicating the HTTP method and URL, and different negotiation and setting formats. The good news is that, while not all security tools support HTTP/2, the tools we are using in this course do all support it. There are still potential problems, however.

Browsers that support HTTP/2 are encrypting the data, precluding us from examining web sessions. Even though the HTTP/2 standard does not require encryption, we have never seen a site running HTTP/2 that isn't encrypted. This means that, unless we are somehow decrypting the data in our conversations, whether or not our tool supports HTTP/2 is moot. We will discuss decryption strategies in Book 4.

HTTP/3 Is on the Horizon

- In November 2018, the IETF agreed to use QUIC as the basis of HTTP/3.
 - We know... We haven't even migrated to HTTP/2 yet either.
- QUIC is another example of efficiency for content delivery networks.
 - Unlike HTTP/2, it will actually speed things up for clients too!
 - You can think of it as HTTP over UDP.
 - Four RFCs of interest:
 - RFC 8999 – Version Independent Properties of QUIC
 - RFC 9000 – QUIC: A UDP-Based Multiplexed and Secure Transport
 - RFC 9001 – Using TLS 1.3 to Secure QUIC
 - RFC 9002 – QUIC Loss Detection and Congestion Control

As if HTTP/2 weren't enough, were you aware that in November 2018, the IETF agreed to use QUIC (Quick UDP Internet Connections) as the basis of HTTP/3? It seems as though protocols are changing more and more quickly as time goes by!

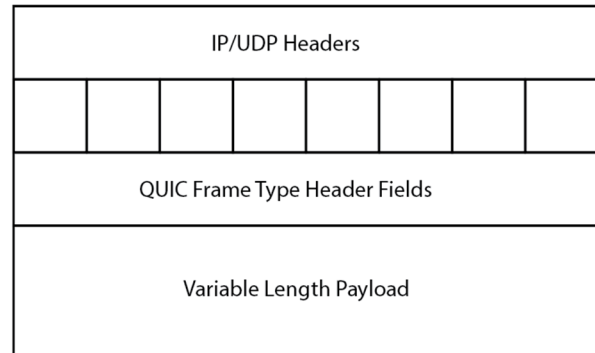
Knowing that it will likely be some time before we are dealing with most sites migrating to HTTP/2, we probably have some time before QUIC really takes hold. While we have certainly run across QUIC (it has been an experimental standard for a few years now) in the wild, it is rare to see. The good news is that, while HTTP/2 particularly optimizes things for the content providers, QUIC will speed things up for the clients too. The key thing that does this is the switch to UDP. This eliminates all the TCP overhead, both in terms of packet header sizes and the constant ACK packets, about half of which contain no data at all.

As of 2021, QUIC has made it to the IETF standards track and now has issued RFC numbers. Don't let this fool you; if you're using Google Chrome and talking to any Google-owned internet property, you have been using GQUIC (Google's version of QUIC) for years.

A Bit About QUIC

Start of QUIC header is an 8-bit frame type

Value	Type	Value	Type
0x00	Padding	0x12-13	Max Streams
0x01	Ping	0x14	Data Blocked
0x02-3	ACK	0x15	Stream Data Blocked
0x04	Reset	0x16-17	Streams Blocked
0x05	Stop Sending	0x18	New Connection
0x06	Crypto	0x19	Retire Connection
0x07	New Token	0x1A	Path Challenge
0x08-F	Stream	0x1B	Path Response
0x10	Max Data	0x1C-1D	Connection Close
0x11	Max Stream Data	0x1E	Handshake Done



QUIC frames come in a variety of types. Overall, the protocol is fairly straightforward, but it can require some effort to decode since the header changes depending on the frame type that you are looking at. We aren't going to take QUIC apart by hand in class. However, you should be in a very good position to be able to read through the QUIC RFC and work out for yourself how QUIC packets are structured and decode them, if need be.

For example, if you peruse the RFC for even a few minutes you will discover that QUIC headers start with an 8-bit frame type value. The remainder of the QUIC header will change depending on the type of QUIC packet you have. Currently, only a small number (relative to the size of the type field) of types are defined, though some variations of QUIC will use some of the bits in this byte for signaling. We're not going to dig into that since those aren't found in the RFC.

What this translates to for us is that we know that, in a QUIC packet, the first byte following the UDP header will be the QUIC type, which we can use to determine what's going on. What we find might tell us that the payload is completely encrypted, so we can't do much more, but even that is useful.

If the data isn't encrypted, we can use the values within the header to begin to reassemble the QUIC stream(s).

Consider One Type

Your packet decoding experience should make this much easier!

- While the RFC isn't short, it is written pretty clearly.
- If you wanted to decode something your tool doesn't understand, you could certainly work it out.

IP/UDP Headers
1 Byte Frame type 0x08 - 0x0F
62 bit Stream ID - Two least significant bits identify origin and direction of the stream
Variable length integer offset Where this chunk goes in the reassembled stream
Stream Data

For example, if we find a frame type between 0x08 and 0x0F, we are looking at a frame containing stream data. Now that we know this is a stream type QUIC header, we can interpret the rest of the header. The very next field is a 64-bit value. The least significant bits are used to indicate who started the stream and which way this data is moving. The high 62-bits are used to identify the stream itself. You can think of this as being equivalent to the IP port pair in a TCP connection, which is used to identify a particular data connection, or stream.

To continue with our analogy of TCP, to determine where a segment of data goes in the reassembled stream, we would use the sequence numbers. In QUIC, a variable-length integer value follows the stream ID. This integer indicates the offset at which this chunk of this stream goes in the reassembled data.

Looking at the frame types, you will see other things that seem familiar: frames for acknowledging data, for establishing a new stream channel over a new IP port pair, terminating a connection, and more.

Why Do We Need This Ability?

QUIC is here today.

- QUIC is currently fragmented different versions; the RFC should change that.
- Support is spotty and often only for specific Google versions.
- Neither Snort nor Suricata support QUIC.
 - Some monitoring tools might support it, but we are unaware of them.
- Wireshark has some GQUIC support.
- Zeek has some GQUIC support through a plugin from Salesforce.
- Security vendors' recommendation today: "Just block it."

QUIC is only growing in popularity!

We're not going to dig any deeper into QUIC, but you should definitely take on QUIC as a personal research project. QUIC is here now, and it's here to stay. Very few network security monitoring tools have any kind of serious support for QUIC. Within Suricata, there is an open feature request to add QUIC. For Snort, we can find no discussion at all of adding in QUIC. Zeek does have support for some QUIC variants as a result of plugins, such as the one from Salesforce, which understands a particular GQUIC dialect.

The general position of the commercial information security tool vendor community is, "Just block it." There are problems with this, however. Blocking it does eliminate decoding as a requirement, or at least it seems to. But could QUIC end up passing over other ports? We've not seen this yet, but since most sites will permit outbound UDP to port 53, we can easily imagine clever people engineering QUIC solutions to use any open port they can find to accomplish QUIC communication. "Just block it" won't work for port 53!

Another big obstacle here is that your QUIC traffic will be encrypted. While it's not a requirement, we have yet to see any real-world QUIC packets that *aren't* encrypted. We will talk more about TLS decryption in Book 4, but as far as we know, there are no solutions, open-source or otherwise, that support decryption of QUIC packets.

HTTP and HTTP Rule Options

Workbook Exercises “HTTP”

This page intentionally left blank.

• An immersion class in speaking packets, protocols, and hex • Ethernet • IPv4 & IPv6 • TCP, UDP, & ICMP • Wireshark & tcpdump fundamentals, filtering, & investigation	Section 2: Signature-Based Detection • Scapy for building tools and simulating signatures & behaviors • Snort & Suricata • Researching common application protocols	• Architecting for massive packet capture & analysis • Overcoming the limitations of signature-based tools • Basic Zeek usage and logs • Threat modelling • Advanced correlation using Zeek	• Guided scenarios to reinforce all skills • Elaboration on effective deployment, collection, and analysis at large scale • Moving beyond alerts to data, driven analysis, & machine learning	• Challenge questions guide/follow expert analysts' progression through real-world data
---	--	---	---	---

- Packet Crafting for IDS/IPS
- Wireshark Part III
- Application Protocols with Snort and Suricata

Signature-Based Detection

Introduction to Snort

Writing Rules

Content Options

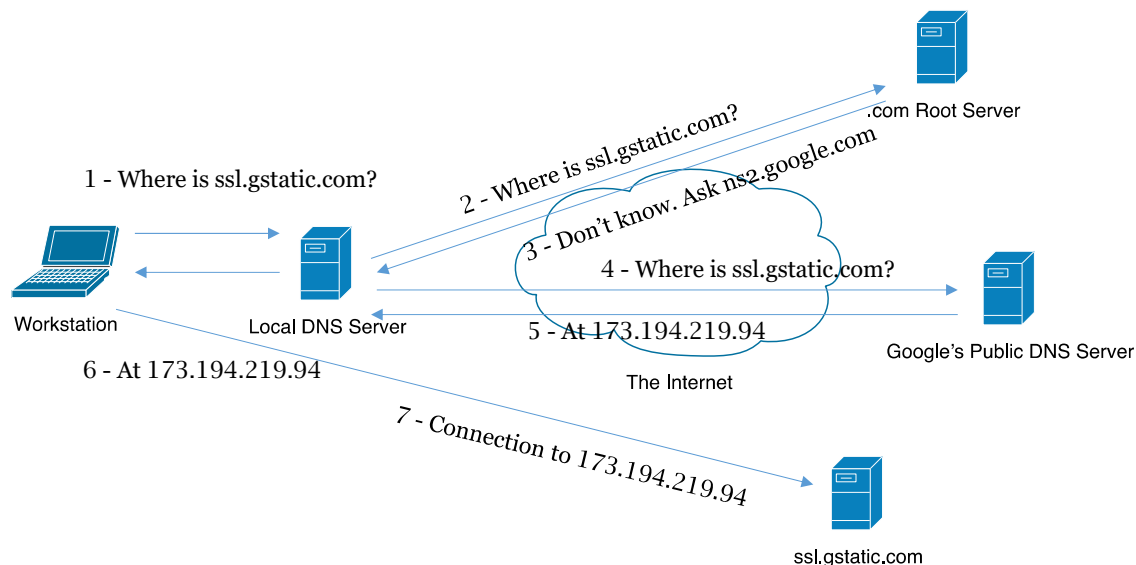
Streams

HTTP and HTTP Rules

DNS and Byte Test

The DNS protocol is ubiquitous. In many ways, it is the base protocol upon which all other application protocols on the internet rely. Surprisingly, Snort doesn't have any real awareness of the DNS protocol, which can lead to rather complex-looking rules using the somewhat opaque **byte_test** option. In this section, we will examine both the protocol and this very useful option.

DNS Resolution: Getting from Here to There



When you actually look at what is involved in resolving a DNS name, it is amazing that the internet is as fast and works as well as it does. From the typical user's perspective, it is as simple as entering a URL into a browser. As we can see here, much more is involved.

Suppose a host wants to go to **ssl.gstatic.com**. To perform the resolution, the resolver library on the local computer has been configured with a local name server. The resolver library generates a forward lookup (translating a name to a number) for the A (or address) record associated with that name. This is sent to the local name server (Step 1).

The local name server, however, has no idea where this host is. The local name server, therefore, sends a request out to the root server for the **.com** zone to find out what to do next (Step 2). Not surprisingly, the root server doesn't actually know the address of the `ssl.gstatic.com` host. It does, however, know where the authoritative server for the **gstatic.com** domain is, `ns2.google.com`, and it tells the local name server so. Since it can foresee that the next question will be "Where's that?" the root server includes the address of the authority server in the response (Step 3).

Armed with this information, the local name server next sends the same query to `ns2.google.com` (Step 4). This server *does* know where that is, so it returns an answer as an A record, `173.194.219.94` (Step 5). Having finally received an answer to the actual question, the local name server sends this response to the workstation (Step 6). Now that the workstation knows where the host is, it can establish a connection to the destination host (Step 7).

If you would like to see how this resolution looks from a packet perspective, feel free to consult the **full-dns.pcap** file with Wireshark or tcpdump.



wireshark full-dns.pcap

Root Server Returns Authority Records

No.	Time	Source	Destination	Protocol	Src Port	Dest Port
1	11:54:41.746782	10.10.10.41	10.10.10.5	DNS	12772	53
2	11:54:41.747576	10.10.10.5	192.41.162.30	DNS	45963	53
3	11:54:41.785989	192.41.162.30	10.10.10.5	DNS	53	45963
4	11:54:41.786551	10.10.10.5	216.239.34.10	DNS	14798	53
5	11:54:41.812121	216.239.34.10	10.10.10.5	DNS	53	14798
6	11:54:41.866065	10.10.10.5	10.10.10.41	DNS	53	12772


```

> ssl.gstatic.com: type A, class IN
  Authoritative nameservers
    ▶ gstatic.com: type NS, class IN, ns ns2.google.com
    ▶ gstatic.com: type NS, class IN, ns ns1.google.com
    ▶ gstatic.com: type NS, class IN, ns ns3.google.com
    ▶ gstatic.com: type NS, class IN, ns ns4.google.com

  Additional records
    ▶ ns2.google.com: type A, class IN, addr 216.239.34.10
    ▶ ns1.google.com: type A, class IN, addr 216.239.32.10
    ▶ ns3.google.com: type A, class IN, addr 216.239.36.10
    ▶ ns4.google.com: type A, class IN, addr 216.239.38.10
  
```

SANS



full-dns.pcap

SEC503: Intrusion Detection In-Depth 117

We're not interested in looking at every packet involved in this exchange right now, but there are a few that are worth a peek. Here we are looking at the response from the root server to the local DNS server. You can see that there are two types of answers being returned.

Within the DNS protocol, **Questions**, **Answers**, **Authority Records**, and **Additional Records** are supported. In this case, the question, if we looked, would be populated with "ssl.gstatic.com". The answers section, however, is empty. This is because this response does not contain an answer to the *actual* question that was asked. The authority section, however, lists all the authoritative name servers that Google has recorded in the **whois** system. The whois system, which operates on port 43, is used to populate the data in the root servers, creating a mapping between the domain name (or zone) and the authority servers.

We can see that the additional records section also has entries. This is where the root server has wisely included the answers to the logical next question: What's the address of ns2.google.com?



To see the output, enter the following on the command line:

wireshark full-dns.pcap

This is record 3.

Local DNS Server Response to 10.10.10.5

No.	Time	Source	Destination	Protocol	Src Port	Dest Port
1	11:54:41.746782	10.10.10.41	10.10.10.5	DNS	12772	53
2	11:54:41.747576	10.10.10.5	192.41.162.30	DNS	45963	53
3	11:54:41.785989	192.41.162.30	10.10.10.5	DNS	53	45963
4	11:54:41.786551	10.10.10.5	216.239.34.10	DNS	14798	53
5	11:54:41.812121	216.239.34.10	10.10.10.5	DNS	53	14798
6	11:54:41.866065	10.10.10.5	10.10.10.41	DNS	53	12772

Domain Name System (response)

[Request In: 1]

[Time: 0.119283000 seconds]

Transaction ID: 0x8b59

Flags: 0x8180 Standard query response, No error

Questions: 1

Answer RRs: 2

Authority RRs: 4

Additional RRs: 4

Queries

▶ ssl.gstatic.com: type A, class IN

Answers

▶ ssl.gstatic.com: type A, class IN, addr 173.194.219.94

▶ ssl.gstatic.com: type A, class IN, addr 173.194.219.120



The last packet that we'll look at that relates to this query is the final answer that is returned from the local name server to the workstation. Notice that this response has entries in all four sections (the Questions, Answers, Authorities, and Additional records sections).

We can see that the query for which this is an answer is "ssl.gstatic.com", looking for an A (address) record. We can also see that there was only one query.

In the next section, we can see that there are two answers. This is confirmed by noting that the "**Answer RRs:**" (RRs stands for Resource Records) line has the number "2" on it, indicating how many answers were returned. In the Answers section, we can see that two address were returned. This indicates that there is some type of load balancing or failover configured.

While they are not visible in the slide, you can also see that the "**Authority RRs**" and "**Additional RRs**" lines indicate four entries each. This lines up perfectly with the authority information that was returned from the root server. In fact, the Google name server returned only the answer records. The local name server has aggregated all of the information that it has discovered and bundled it all into this response.



To see the output, enter the following on the command line:

wireshark full-dns.pcap

This is record 6.

Sample tcpdump DNS Query/Response

The diagram shows two lines of tcpdump output. The first line is a query from 192.168.11.14 to 192.168.11.1:53 with transaction ID 30644, flags 0x00000001, and query type A for www.sans.org. The second line is a response from 192.168.11.1 to 192.168.11.14:27383 with transaction ID 30644, flags 0x00000000, and one A record for 66.35.59.202. Annotations with arrows point to specific fields: 'Query ID' points to 30644 in both lines; 'Recursive' points to the 1 in the query flags; 'Address' points to the IP 192.168.11.1 in the query; 'This is a query' points to the query type A; 'Answers' points to the 1 in the response flags; 'Authorities' points to the 0 in the response flags; 'Additional' points to the 0 in the response flags; and 'The answer contains A records' points to the A record in the response.

```

192.168.11.14.27383 > 192.168.11.1.53: 30644+ A? www.sans.org. (30)
192.168.11.1.53 > 192.168.11.14.27383: 30644 1/0/0 A 66.35.59.202
(46)
  
```



Let's interpret the tcpdump output for DNS: 30644 is the DNS transaction ID (or Query ID) number. This is a 16-bit value that is used to pair requests and responses. We'll discuss the identification field in more detail when we look at DNS cache poisoning. The "A" notation signifies that this is an address lookup. The plus sign is the way that tcpdump conveys that the DNS flags field contains a bit setting of 1 for "recursion desired."

Recursion tells the DNS server that, if it doesn't know the answer, it should go and find out by asking the upstream or root name servers. Some DNS servers will not perform recursion, notably root servers or top-level domain DNS servers. They are busy machines and cannot, or more accurately will not, process recursive queries. These high-level DNS servers are expected to give only authority information for a domain.

Our perspective is limited to what occurs on our local network only. It is possible that 192.168.11.1 contacts a root server for a referral for the authoritative server for www.sans.org, contacts a generic top-level domain server, contacts an authoritative server for www.sans.org directly, or has the IP cached. Regardless, we see it return a response with a DNS transaction ID of 30644 and an address of 66.35.59.202.

The 1/0/0 notation means that one answer record was returned (the IP address) and no authoritative or additional records were returned. As a reminder, a *resource record* is DNS terminology for a DNS record consisting of a DNS name, type, class, and potentially other information, such as the TTL, queries, responses, authority records, and additional records.

Authority records contain data on the "owning" domain. An additional resource record contains more information to assist in the resolution. For instance, it may give the IP address of the authority records returned. This is efficient because the receiving DNS server does not have to perform another query to find this data.



To see the output, enter the following on the command line:

```
tcpdump -ntr sans-dns.pcap
```

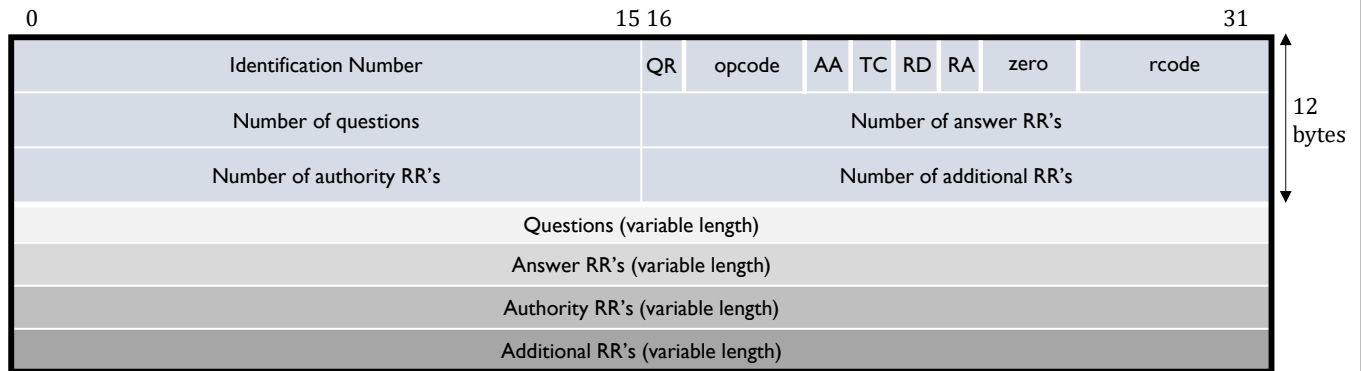
Common DNS Resource Record Types

Type	Function
A	Address, returns 32-bit IPv4 address
AAAA	Address, returns 128-bit IPv6 address
CNAME	Canonical name, alias for name
MX	Mail exchange, name of mail server
NS	Name server, authoritative name servers for zone
PTR	Pointer, reverse DNS lookup
SOA	Start of authority, authoritative information about zone

DNS records may be referred to by an abbreviated type. The most common ones are listed. You are probably most familiar with the *A* type or IPv4 address record(s) returned for a given DNS name resolution. The *AAAA* (*quad-A*) is the same thing except it's for IPv6 address records. The *CNAME* (*canonical name*) is an alias for a given DNS name. For instance, this may be used if a single server hosts many different websites or if a single server has multiple purposes like **www.example.com** and **ftp.example.com**.

The *MX* mail exchange record identifies the mail server for the domain. The *NS* (*name server*) record holds the name of an authoritative name server for the domain. There may be multiple NS records for a given zone. The *PTR* (*pointer*) record performs a reverse lookup when an IP address is known and the domain name is wanted. Finally, the *SOA* (*start of authority*) record gives information about a particular zone.

Format of DNS Data



We're going in for a deep dive of decoding the DNS data format because DNS is such an important protocol that is often used, and this will facilitate understanding some examples to come. Also, DNS can be used for tunneling data because it is very unlikely that there are firewall rules blocking it from exit and entry. Knowing the DNS format will help you examine the protocol to inspect it to discover if it is normal or not.

We start with the static portion of the DNS record. Each DNS query or response has an identification or transaction number that is used to pair requests and responses. There are several DNS flags. The first is the high-order bit that signifies a query (value 0) or response (value 1), a 4-bit opcode that is mostly obsolete, a 1-bit response flag denoting whether or not the answering DNS server is authoritative, a 1-bit truncation code in a response that indicates that the query must be reissued using TCP instead of UDP, a 1-bit recursion desired for the query, a 1-bit recursion available for the response, 3 bits that are reserved or not used (though, 2 of these bits have been put to use in DNSSEC to mark "Authentic Data" and "Checking Disabled"), and a 4-bit return code in the reply reflecting whether or not the query could be answered.

Next, there are counts for the number of queries; it should always be 1, but there is nothing in the protocol that prevents it from being greater. This is followed by the number of response, authority, and additional resource records.

The variable part follows. The data in a given resource record is variable in length. The resource record itself contains this length. For instance, the length of a hostname in the query can be different. Another variable part of the DNS format is that there can be more than one occurrence of a resource record. We will see that DNSSEC often contains multiple answer resource records.

Format of Answer Resource Record (I)

0x0000:	4500	0098	0000	4000	4011	a2ee	c0a8	0b01	E.....@. @.....
0x0010:	c0a8	0b15	0035	bfb6	0084	d1bd	d36d	8180	5.....m..
0x0020:	0001	0006	0000	0000	0661	6d61	7a6f	6e03	amazon.
0x0030:	636	6d00	0001	0001	c00c	0001	0001	0000	com.....
0x0040:	000c	0004	36ef	19d0	c00c	0001	0001	0000	6.....
0x0050:	000c	0004	36ef	1107	c00c	0001	0001	0000	6.....
0x0060:	000c	0004	36ef	19c8	c00c	0001	0001	0000	6.....
0x0070:	000c	0004	36ef	1a80	c00c	0001	0001	0000	6.....
0x0080:	000c	0004	36ef	19c0	c00c	0001	0001	0000	6.....
0x0090:	000c	0004	36ef	1106					6...
Query ID	Flags	Queries	Answers	Authorities	Additional				

I am a label: 6 bytes

I am a label: 3 bytes

End of name

Class 1 (IN)

06	61	6d	61	7a	6f	6e	03	63	6f	6d	00	00	01	00	01
amazon						com			Type 1 (A)						

SANS



dns-amazon.pcap

SEC503: Intrusion Detection In-Depth 122

It's worth taking a few minutes to examine how DNS records are encoded. On the slide, you can see that the IP header has been shaded in one color and the UDP header has been shaded in another color. Following that, we have called out the DNS header, highlighting the query ID, DNS flags, number of queries, number of answers, number of authority records, and the number of additional records. These parts of the header are static and will always be present.

How you interpret the rest of the DNS packet or message depends on the values found in queries, answers, authorities, and additional records. In this case, queries and answers both have a value greater than zero, so there are queries and answers present.

Continuing on with our decoding, the next value circled is the 0x06 that follows the number of additional records field. This field, known as a DNS Label, indicates the number of bytes that follow. These bytes make up the first portion of the DNS name. In this case, 0x06 is followed by the letters 'amazon.' Following this, we find a 0x03. This is another label, this time indicating three bytes or characters. The characters that follow are 'com,' after which we find the value 0x00, which indicates the end of the DNS name.

That's no big deal, but something should jump out at you. Writing a rule that reads 'content:"amazon.com"' that watches for DNS queries or answers involving amazon will *never* alert because there is no dot in that name within the DNS message! OK, so that's useful, but there's more, and the "more" is unpleasant.



Format of Answer Resource Record (2)

0x0000:	4500	0098	0000	4000	4011	a2ee	c0a8	0b01	E.....@. @.....
0x0010:	c0a8	0b15	0035	bfb6	0084	d1bd	d36d	8180	5.....m..
0x0020:	0001	0006	0000	0000	0661	6d61	7a6f	6e03	amazon.
0x0030:	636f	6d00	0001	0001	c00c	0001	0001	0000	com.....
0x0040:	000c	0004	36ef	19d0	c00c	0001	0001	0000	6.....
0x0050:	000c	0004	36ef	1107	c00c	0001	0001	0000	6.....
0x0060:	000c	0004	36ef	19c8	c00c	0001	0001	0000	6.....
0x0070:	000c	0004	36ef	1a80	c00c	0001	0001	0000	6.....
0x0080:	000c	0004	36ef	19c0	c00c	0001	0001	0000	6.....
0x0090:	000c	0004	36ef	1106					6...

6 bytes, amazon

3 bytes, com

Type and Class

C00C 0001 0001



I am a pointer

To offset 0x000c

SANS



dns-amazon.pcap

SEC503: Intrusion Detection In-Depth 123

Look at what comes immediately after the type and class values, each of which are 0x0001. We have called those bytes out to the bottom of the slide: 0xC00C 0001 0001. What does this mean? Normally, we would expect to find a label, length, or field, but each portion of a DNS name is limited to 64 bytes, so 0xC0 clearly cannot be a length. Instead, when those two high-order bits are turned on, the meaning changes. Now the next 14 bits represent a pointer to where the DNS name can be found. When we decode those 14 bits, we find that they contain 0x000C, or 12. Twelve bytes from where? From the beginning of the DNS header!

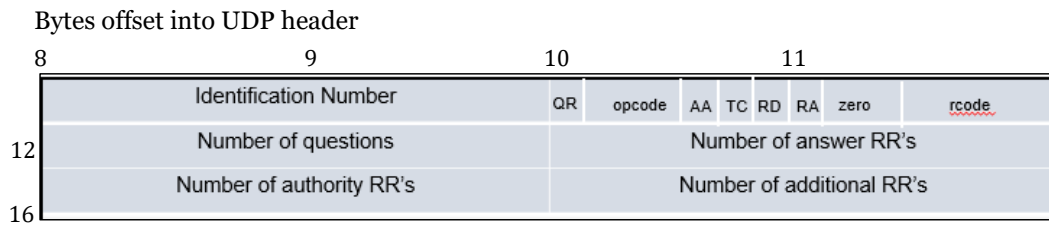
This means that the DNS name that should be used in the first answer record *points to* the "amazon.com" from the query record. In fact, all six of the answers in the DNS packet point to the same place!

We're not going to make you decode any of these, but this is pretty important. DNS does this in an attempt to be efficient, and it works well. However, it means that if someone wanted to obfuscate a DNS query or response, it is pretty easy to do so. Taking our example of "amazon.com" again, the following content rule would not be reliable:

```
content:"|06|amazon|03|com";
```

Why not? Because there is no requirement that those two parts of the name be next to each other. Not only this, but they could actually appear in any order in the message!

BPF to Extract DNS Records



<code>tcpdump -r dns-bpf.pcap -nt 'dst port 53 and udp[10] & 0x80 = 0'</code>	All queries
<code>tcpdump -r dns-bpf.pcap -nt 'src port 53 and udp[10] & 0x80 = 0x80'</code>	All responses
<code>tcpdump -r dns-bpf.pcap -nt 'src port 53 and udp[10] & 0x82 = 0x82'</code>	Response with TC
<code>tcpdump -r dns-bpf.pcap -nt 'dst port 53 and udp[10] & 0x81 = 0x01'</code>	Query with RD
<code>tcpdump -r dns-bpf.pcap -nt 'src port 53 and udp[10:2] & 0x8180 = 0x8180'</code>	Response RD + RA



Remember your dear friends the Berkeley Packet Filters? They can be applied if you have a large number of records in a pcap as an efficient way to search for particular static DNS header characteristics.

You can find normal traffic such as all queries or all responses. Or you can look for anomalous characteristics in the fixed part of the DNS header. Once you advance into the packet into the DNS resource records, the fields become variable, possibly in length and number of resource records. You need a true DNS decoder such as found in Wireshark or Zeek to extract records based on content in the DNS resource records.

The starting point for examining DNS content using BPF is the beginning of the UDP record. Unfortunately, the filter language does not offer DNS as the starting point. It's really not much of a change; you must start beginning at 8 bytes offset after the static size UDP header.

You see some filters in the slide used to find some normal traffic. You will get a chance to find anomalous DNS characteristics in the exercises that accompany this section. The first filter requires a destination port of 53 and examines the high-order bit value of offset 10, where the flag indicates 0 for a query. The second filter requires a source port of 53 and examines the high-order bit value of offset 10 for a flag bit value of 1 to signify a response. The third filter also looks for a response, yet in conjunction with the truncation bit set.

The fourth filter looks for a query with the RD bit set. We must test the query bit to make sure it is not set, and yet the RD flag in the low-order bit of offset 10 must be set. The fifth filter requires examining both bytes 10 and 11 because it tests for a response bit value of 1, the RD flag set in the low-order bit of offset 10, and the high-order bit of offset 11 set for the RA flag.

DNS Zone Transfer

No.	Time	Source	Destination	Protocol	Src Port	Dest Port	Info
6	0.000040	172.16.16.164	172.16.16.139	DNS	1108	53	Standard query 0x0007 AXFR contoso.local
7	0.021769	172.16.16.139	172.16.16.164	DNS	53	1108	Standard query response 0x0007 SOA dns3.contoso.local A

▼ Answers

- ▶ contoso.local: type SOA, class IN, mname dns3.contoso.local
- ▶ contoso.local: type A, class IN, addr 172.16.16.139
- ▶ contoso.local: type NS, class IN, ns dns3.contoso.local
- ▶ _msdcs.contoso.local: type NS, class IN, ns csanders-9ceae1.contoso.local
- ▶ _gc._tcp.Default-First-Site-Name._sites.contoso.local: type SRV, class IN, priority 0, weight 100, port 3268, target dns3.contoso.local
- ▶ _kerberos._tcp.Default-First-Site-Name._sites.contoso.local: type SRV, class IN, priority 0, weight 100, port 88, target dns3.contoso.local
- ▶ _ldap._tcp.Default-First-Site-Name._sites.contoso.local: type SRV, class IN, priority 0, weight 100, port 389, target dns3.contoso.local
- ▶ _gc._tcp.contoso.local: type SRV, class IN, priority 0, weight 100, port 3268, target dns3.contoso.local
- ▶ _kerberos._tcp.contoso.local: type SRV, class IN, priority 0, weight 100, port 88, target dns3.contoso.local

Zone transfers typically occur over TCP, just like queries that result in large answers.

SANS



dns-axfr.pcap

SEC503: Intrusion Detection In-Depth 125

Pictured in the slide is a zone transfer requested by 172.16.16.164. This, presumably, is the secondary DNS server. The primary DNS server is 172.16.16.139. As you can see, the session is transported over TCP. The secondary server requests a *zone transfer (AXFR)* for the local domain **contoso.local**. The primary server responds by sending the client all the domain records.

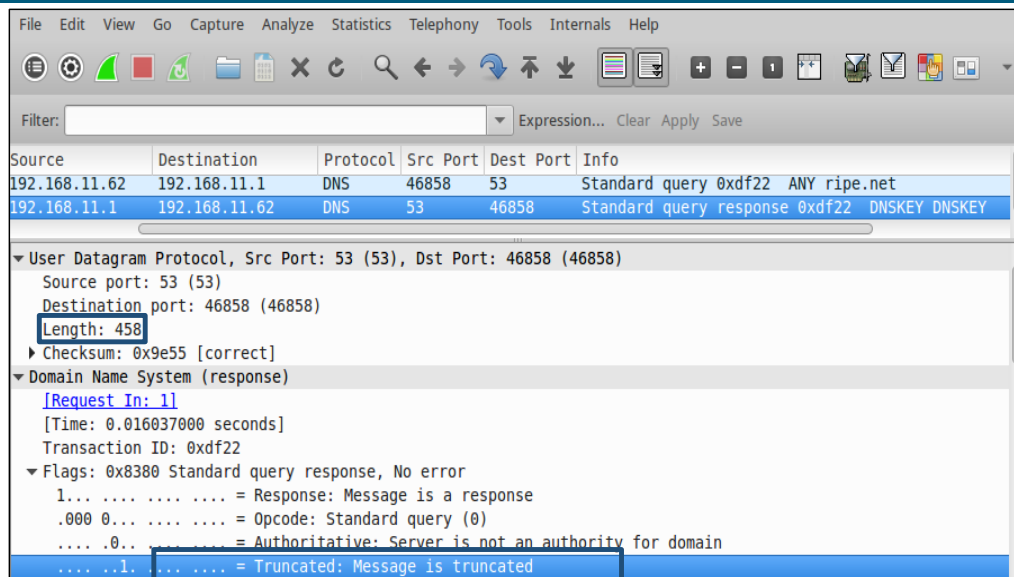
Special thanks and attribution to Chris Sanders for this pcap.



To see this output, enter the following on the command line:

wireshark dns-axfr.pcap

Large DNS Response



SANS



longdns.pcap

SEC503: Intrusion Detection In-Depth 126

Some issues are associated with the transport protocol for DNS traffic. Typically, DNS queries are sent and received using UDP because answers are often succinct, and hosts can tolerate a best delivery effort because they can reissue DNS queries.

In the past, the maximum allowable size for a UDP DNS payload was 512 bytes. This was prior to the use of EDNS, an extension mechanism for DNS that expanded the size. What happens if a server does not support EDNS and the DNS response exceeds 512 bytes? The first thing that occurs is that the UDP response in record 2 is returned with the truncated bit located in the DNS flags turned on, as displayed in the DNS flags field highlighted in the rectangle at the bottom in the above slide. The response has 450 bytes of data after discounting for the 8 bytes of the UDP header included in the UDP length.

This doesn't exceed 512 bytes, but the 450 bytes contain entire resource records. In other words, the response was truncated after the last resource record that could be wholly contained within the 512 bytes. In fact, if you expand the UDP response in Wireshark's packet detail pane, you see that the DNS message ends with a DNSKEY resource record.

192.168.11.62 receives the response with the truncation bit set and reissues the DNS query using TCP. If you were to examine the rest of the pcap file, you would see the query is reissued in TCP and a full response is received.



To see this output, enter the following on the command line:

wireshark longdns.pcap

DNSSEC

- Ensures integrity and authenticity of DNS record(s)
- Uses public key cryptography to digitally sign responses
- Introduces some new DNS resource records

Type	Function
RRSIG	Signature for a resource record set
DNSKEY	Public key for DNS zone
DS	Signing key of a delegated zone
NSEC	Proof that a name does not exist

Domain Name System Security Extensions (DNSSEC) was introduced in an attempt to provide some security measures for DNS records. Specifically, DNSSEC attempts to ensure that the source of a DNS response is from an authorized source. Sites concerned with security have implemented DNSSEC. However, the global security of DNS requires the cooperation of all DNS sites and requires a reliable and secure infrastructure to disseminate public keys for DNS sites or zones. Universal adoption of it has been slow and, therefore, incomplete as a security enhancement.

DNSSEC does not attempt to provide confidentiality of queries or responses. There are many aspects to DNSSEC, such as performing key distribution, data origin authentication, and transaction and request authentication. Our discussion concentrates only on data origin authentication, but RFC 2535 discusses in complete detail all the other features included in DNSSEC.

To ensure the integrity and authenticity, DNSSEC has added some new DNS resource records. The *RRSIG* record contains a cryptographic hash of the original resource record that is signed using the private key belonging to the true owner of the records. The *DNSKEY* record is the public key of the signer of the records to be used by the recipient of DNS records to validate the authenticity. The *DS* record represents a *DNSKEY* signed by the parent zone administrators to establish a chain of trust.

Finally, the *NSEC*, or *NextSECure*, record is used to prove a name does not exist. For instance, suppose the name of your DNS zone is *whatever.com* and you have three address records: **dog.whatever.com**, **monkey.whatever.com**, and **zebra.whatever.com**. Say someone asks for an address resolution of *emu.whatever.com*. An *NXDomain* (*nonexistent*) response is returned to indicate that **emu.whatever.com** is unknown.

Can that response be trusted? It can be using a scheme in which an *NSEC* record exists for all next address records. In other words, *dog.whatever.com* has an *NSEC* record of **monkey.whatever.com** that has an *NSEC* record of **zebra.whatever.com**. No other DNS type A record falls between these pointers; therefore, resolution of **emu.whatever.com** should return a signed *NXDomain* response. This is like a linked list with chronological values of the resource names. We'll see where this has applicability when we cover the Kaminsky cache poisoning attack.

Extended DNS (EDNS)

No.	Source	Destination	Src Port	Dest Port	Info
1	10.5.1.12	10.5.1.119	56725	53	Standard query 0x4341 A dshield.org
2	2601:340:81::	2001:500:3::42	18772	53	Standard query 0x4de0 A dshield.org

Domain Name System (query)
[Response In: 14]
Transaction ID: 0x4341
Flags: 0x0100 Standard query
Questions: 1
Answer RRs: 0
Authority RRs: 0
Additional RRs: 1
Queries
Additional records
 <Root>: type OPT
 Name: <Root>
 Type: OPT (EDNS0 option)
 UDP payload size: 4096
 Higher bits in extended RCODE: 0x0
 EDNS0 version: 0
 Z: 0x8000
 Bit 0 (DO bit): 1 (Accepts DNSSEC security RRs)
 Bits 1-15: 0x0 (reserved)
 Data length: 0

We can resend this same message three times before we send the same number of bytes as a UDP query followed up by a change to TCP!

SANS



dnssec-dshield.pcap

SEC503: Intrusion Detection In-Depth 128

Before we move on, we need to discuss *Extended DNS*, or *EDNS*, which was created to deal with the 512-byte maximum limit for DNS data. EDNS was used in all the DNSSEC records we examined, but it isn't limited to DNSSEC. You will find it is used by default by Windows 10 and higher, and that it is quite useful when we have a single domain name returning a dozen or more addresses.

EDNS uses what it calls a *pseudo resource record*, which is placed in the additional resource records section of a query or response. As you are aware, DNS records have types such as IPv4 address, IPv6 address, and name server, to mention a few. The EDNS pseudo resource record has a type of OPT denoted by a value of 0x00 29.

Look at the query from packet 1 in **dnssec-dshield.pcap** in the Wireshark packet details pane. A typical query has a query resource record only and a count of 1 in the number of questions. EDNS uses the additional resource record for the pseudo resource record content and therefore has a count of 1 in the additional resource record number.

There are some fields that are not present in a normal DNS resource record. These include the UDP payload size, the EDNS version and EDNS data, an extended return code, and the Z field used to express support for the use of DNSSEC. The high-order bit value of the Z field indicates that DNSSEC is supported when set to 1 and not supported when set to 0.



To see this output, enter the following on the command line:

wireshark dnssec-dshield.pcap

This is record 1.

IPv6 DNS

Changes needed, IP address 128 bits, not 32 bits

New resource record type AAAA (quad-A):

- Same as address record type (A), but four times larger

New reverse lookup top-level domain:

- ip6.arpa

Needs new server and client resolution software to support changes

Changes in DNS are required to deal with the IPv6 128-bit address scheme. A new record type, known as the quad-A type, was added because it is four times larger than an IPv4 address A type. Also, there must be a new top-level domain to handle reverse lookups; this is **ip6.arpa**.

As far as software is concerned, you need DNS server software to support the new record types, and there must be built-in support for IPv6 client resolution for any host doing DNS lookups.

How do DNS servers deal with IPv4 and IPv6 at the same time? First, they can store both IPv4 and IPv6 records if the software supports IPv6. IPv6 records are typically added to a DNS server that already has IPv4 records.

It is important to understand that it is not necessary for a DNS server to actually communicate over IPv6 to answer a request for an IPv6 address. It is just resolving some kind of query and can be transported over IPv4. Finally, recursive servers that actually find the answer to a query may have to support dual IPv4 and IPv6 stacks for a long time because the change to IPv6 will be gradual, and these recursive servers may be communicating with both IPv6 and IPv4 stacks.

DNS Attacks

Fast-flux

- Commonly used for C2 and malicious 2nd stage payloads
- Rapidly changing resolution using short TTLs for records

DNS spoofing

- MITM attack, forging answers
- Captive portals, ISPs, and anyone who can see your DNS queries

DNS cache poisoning

- Essentially, DNS spoofing
- Goal is to poison an authority record for a domain in a resolver

In the past, when a compromised host was discovered to be associated with malicious activity, it could be blocked using the IP address of the host. Hackers and criminals who now have botnet armies at their disposal can use any of the many hundreds or thousands of hosts for the purpose of supporting malicious activity. If only the hackers could associate many different IP addresses of the botnet hosts with a given hostname, they could make it difficult to identify and block the activity.

A scheme has been concocted known as *fast-flux* DNS that uses an ever-changing set of IP addresses associated with a particular hostname to elude blocking and identification. This is accomplished by associating multiple address records (A records) with a given hostname. In itself, this is not unusual or malicious. If you do an address resolution of **www.google.com**, you can find several IP addresses associated with the hostname for the legitimate purpose of having multiple different hosts serving up the same content.

What makes fast-flux different is that the address records are given a low DNS Time to Live (TTL) value so that they expire quickly, and new address records are assigned to the same hostname. This provides a constantly changing set of IP addresses, likely those of other botnet hosts directly hosting or redirecting the malicious activity. This makes the malicious hosts very difficult to block or identify.

This is known as *single-flux* DNS manipulation. There is another scheme called *double-flux* where both the address records and the name server records for the authoritative name servers for the malicious domain change, too. This makes the activity more difficult to block because blocking the authoritative name server activity becomes untenable as it too is changing along with the address records.

A simple attack against the DNS system is something much more "local." If an attacker is either on the local segment or sits in a location between your DNS server and the upstream DNS server, she has the opportunity to sniff the DNS query. Doing so, she has access to everything needed to generate a false response. For a response to be accepted, putting DNSSEC aside, the response must come from the correct IP address and the correct port number and have the correct query ID. As long as these elements match, the DNS response will be accepted.

Will the real response, which arrives later, overwrite this spoofed response? No! Why not? Think not about how DNS works and think about how UDP works. When a query is sent, an ephemeral port is allocated, and the request is sent from it. When an answer arrives on that port, what happens? The application that sent the request *stops listening*. This actually provides a potential way to detect if this type of activity is occurring; what if we were to watch for hosts in our network sending back port unreachable messages to DNS responses? Of course, we would need to differentiate between random packets coming in to port 53 and packets that were legitimate responses to queries. Still, if we did so, this could provide a possible detection mechanism for this type of attack.

Of course, while a malicious attacker can certainly use this technique to redirect local users, we also find that organizations viewed as more "legitimate" are also using this technique. Some captive portals and many AUP systems (appropriate use products) work by spoofing DNS responses, preventing users from getting places. Additionally, some ISPs, particularly CenturyLink, prevent their customers from sending queries to any DNS servers but the CenturyLink servers. Why? Because this allows them to potentially do filtering or, far more attractive for a corporation, to harvest information about where you go on the internet to sell to advertisers and marketers!

To explain how cache poisoning can still be achieved today, it's useful to understand how it used to work in the old days. Attackers realized that they could leverage the efficiencies of the DNS system, sending both queries and responses in the same packet request.

Consider the packet in the slide. We see a DNS query being sent to 172.16.2.54 that contains a request for "www.sans.org." However, notice that there are also authority and additional records. What's really interesting is that the authority record *has nothing to do* with sans.org!

Looking at this today, you might be surprised to learn that this was an extremely effective cache poisoning attack, convincing a name server to cache an invalid authority record for the bankofamerica.com domain, in this case. Originally, DNS servers were very trusting. Even though the authority information has nothing to do with the query, the server would happily cache this data.

These days, not only will the server *not* accept that data on a query, a feature called *bailiwick checking* was added, preventing a server from accepting data for an unrelated domain. (Bailiwick is an antiquated English word referring to an "area of responsibility.") Elements of this attack, however, are still used in modern DNS poisoning attacks.

DNS Is Changing

DNSSEC has been around for a long time with very slow adoption

Some efforts to move to DNS over TLS

- Cloudflare offers service at 1.1.1.1
- DNS over TLS operates over port 853

Significant efforts to shift to DNS over HTTPS

- Governments and enterprises are rightfully concerned!

What does this mean for us?

- Encrypting DNS is great for personal users on personal devices.
- Not great in a corporate environment
- Monitor for signs of alternate DNS resolution paths (more in Book 4!).

While DNS has remained largely unchanged for decades, there have been innovations over the years, including the attempted introduction of DNSSEC and Extended DNS. More recently, a new way of operating DNS has come about. Sometime around 2014, experimentation began on performing DNS queries over TLS. Recently, this has become an experimental/draft standard. Unlike some of the other efforts made to improve DNS over the years, this is being championed by a large organization with a great deal of "weight" on the internet—CloudFlare. In fact, CloudFlare has made its DNS over TLS service available for free public use at address 1.1.1.1.

From a privacy perspective, this is great. It will be much more difficult for ISPs and others to snoop on your queries to figure out what you're doing. This is also great because, potentially, it would eliminate spoofed responses, as long as we require a valid certificate that matches the zone for which the response is sent. For enterprise defenders, however, this news is less great. *Many* of our security products rely on the ability to determine which hosts you are looking up via DNS.

Our current recommendation is that, while this is wonderful for personal use, enterprise networks should likely be configured to block outbound connections on port 853 (DNS over TLS operates over port 853). Instead, all outbound queries should be forced to go through your official DNS resolution path, providing you at least some insights into how the network is being used.

Certainly, there can be legal ramifications to this, and we encourage you to investigate what is and is not legal in your jurisdiction and on the networks that you monitor. However, in our experience, if you are simply tracking the lookups and not *who* is looking up what, you are generally in the clear. Of course, we are not lawyers, and we are not the ones who would be held liable if you ran afoul of a law or regulation, so verify what you can and cannot do with your legal department.

Snort Needs a Real DNS Protocol Decoder

0		1		2		3					
DNS identification number				Q R	opcode	A A	T C	R D	R A	zero	rcode

```

alert udp $EXTERNAL_NET 53 -> $HOME_NET any\
(msg:"DNS large number of NXDOMAIN replies - possible DNS cache\
poisoning"; byte_test:1,&,2,3; byte_test:1,&,1,3;\
byte_test:1,&,128,2: etc.)

```

Current Snort detection:	byte_test:1,&,128,2	byte_test:1,&,2,3 byte_test:1,&,1,3
What if Snort had a DNS decoder that allowed you to test DNS fields?	dns_type = 1	dns_rcode = 3

A lot of DNS responses within a short amount of time containing NXDOMAIN (nonexistent domain) could be a sign of cache poisoning. The rather confusing Snort rule seen on this slide can detect cache poisoning by finding a large number of DNS responses that contained a return code of 3, meaning that there is a name error or nonexistent domain.

The pictured rule extracts bits using its own brand of bit masking and test. We don't expect to see such primitive processing in a Snort rule because Snort has preprocessors that parse most major protocols. Unfortunately, Snort does not have a true DNS decoder.

This does not reflect well on the software because it cannot decode the DNS records like a DNS server does. Hence, it would have trouble following pointer logic or trouble examining any field other than the static ones at the beginning of the DNS header.

Snort needs a DNS decoder that makes parsed DNS fields available to the user in a rule. If the DNS flags field was parsed and available as a keyword, no bit masking and Boolean logic would be required. And you could actually examine values in resource records if the capability were available. DNS detection is incomplete and difficult using Snort.

Snort's `byte_test` Option

How do you deal with protocols that Snort doesn't decode?

- Your ability to read and decode packets helps a great deal.
- Coupled with the patience to read and understand an RFC, you can decode fields.

Snort provides the `byte_test` (and related) options:

```
byte_test: #bytes, operation, value, offset
```

There are many other optional parameters to refine this.

- We will experiment with this a bit in the next lab!

While it is unfortunate that Snort doesn't provide a protocol-level decoder that adds content and other options to the rules language, we can overcome this deficiency. In fact, the mechanism we can use for this can be used to solve this problem for DNS and can be used more generally to analyze *any* protocol!

The **`byte_test`** option allows you to zero in on a byte or set of bytes anywhere in the packet payload and test it for various conditions. The first parameter is the **number of bytes to operate on**. The maximum allowable number is 10, though when used with the MSRPC decoder, you are limited to 1, 2, or 4.

The second parameter is the **operation to perform** on those bytes. This can be any of `<`, `>`, `<=`, `>=`, `=`, `&`, and `^`. All of those are intuitive in terms of their meaning, except for the last two. The `&` parameter is, as you might expect, a bitwise AND, just as we used it in BPF bit masking. The final option, `^`, is *not* XOR, which is what most with a programming background would expect; instead, it is a bitwise OR.

The third parameter is the **value to compare to/operate on** relative to the bytes that were selected. In other words, if we used the `>` operator, we would be checking if the bytes in the packet are greater than the value provided. On the other hand, if we used the `&` operator, this would apply the value provided as a bitwise mask against the bytes in the packet. The result is used as a true/false value to determine if this rule matches the packet.

The fourth parameter is the **offset within the payload** from which the bytes should be extracted. There are a number of additional options that can be used to change how the bytes are processed, converting the data to other formats or possibly the endianness of the data. Related options include **`byte_jump`**, **`byte_math`**, and **`byte_extract`**.

DNS Review

Host-to-IP resolution must be done before any IP traffic can be sent.

Different ways to discover information about DNS servers and associated hosts

DNS responses untrustworthy unless DNSSEC used

DNS is a core infrastructure protocol; flaws can allow attackers to redirect traffic.

Best examined with IDS/IPS protocol decoder

Let's wrap up what we've covered in the DNS section. First, we discussed the need for translation from hostname to IP address and other resolutions. DNS transactions are necessarily transparent because the intent is to disseminate data, enabling attacks to perform some reconnaissance.

Remember that DNS responses are untrustworthy because you cannot be sure that an authoritative or legitimate DNS server responds to a query unless DNSSEC is used. And it is worth repeating that DNS is flawed and remains a weak link and a continuing hazard as a core infrastructure protocol.

Because DNS is a protocol that uses variable-sized records and has some unique ways of specifying DNS data via the use of pointers, it is not an easy protocol to parse using simple fixed offsets/values or even pattern matching. It is best examined with a protocol decoder that can scrutinize individual field and value pairings.

DNS and Byte Test

Workbook exercises "DNS"

This page intentionally left blank.



Section 3: Bootcamp

This page intentionally left blank.

• An immersion class in speaking packets, protocols, and hex • Ethernet • IPv4 & IPv6 • TCP, UDP, & ICMP • Wireshark & tcpdump fundamentals, filtering, & investigation	• Scapy for building tools and simulating signatures & behaviors • Snort & Suricata • Researching common application protocols	• Architecting for massive packet capture & analysis • Overcoming the limitations of signature-based tools • Basic Zeek usage and logs • Threat modelling • Advanced correlation using Zeek	• Guided scenarios to reinforce all skills • Elaboration on effective deployment, collection, and analysis at large scale • Moving beyond alerts to data, driven analysis, & machine learning	• Challenge questions guide/follow expert analysts' progression through real-world data
---	--	---	---	---

Bootcamp (Sections 1-5): Real-World Application of Theory

• Bootcamp

Section 3 Bootcamp

Processing Snort Alerts

Researching Protocols

Researching SMB

Snort/FirePOWER/Suricata Logs

- You've run your IDS and produced some logs... Now what?
 - What's important?
 - What's marked as low priority that might actually matter for you?
 - Why did that rule fire anyway?
 - What does that rule do?
- In other words, what am I supposed to do with all of these alerts?



The first and more important part of this book is focused on how to deal with the alerts that are being generated by your IDS. There's clearly a lot there, but which alerts matter? Is it the alerts that happen the most often? Is it the alerts with the highest priority? How can you zero in on which alerts matter? How can you identify which systems are most likely to have been compromised?

These questions, and others, form the basis for IDS log analysis. Certainly, these logs can be forwarded into or imported into your organization's SIEM. Still, putting them into the SIEM doesn't help if you still don't know what matters and what doesn't! Also, your SIEM will not always allow you to import the packet data that is generated in the snort.log that accompanies the alert log that is generated. So, what are we to do?

Processing Logs

- Start by answering some basic questions:
 - Which alerts appear the most often?
 - Of those alerts, which are the highest priority?
 - Looking at the systems involved, which systems have the most associated alerts?
- How can we answer these questions?

Here are three big questions that are always good to answer:

Which alerts appear the most often? This question is useful to answer because it can indicate an issue on your network that malware or attackers can see that you have not yet mitigated. Even if they are not successfully exploiting the issue, it would be wise to resolve whatever configuration makes this an attractive target for attackers.

Of those alerts, which are the highest priority? It is rare that we have only a few alerts to worry about. Typically, we have hundreds or thousands of alerts that are firing. Once we figure out which alerts are happening frequently, the next step in triage is to figure out which matter the most. For example, imagine that there are 400,000 attempts to log in to an FTP account but only one successful buffer overflow attack.... The 400,000 login attempts are clearly more frequent, but an attempted access is a lower priority than a successful access; even though that happened only one time; clearly, we should evaluate that threat first!

Looking at the systems involved, which ones have the most associated alerts? Another interesting view of your data is examining which of your hosts are most frequently involved in alerts. Similarly, identifying which external systems are most frequently triggering alerts is also useful. For external hosts, this might lead to a decision to block that source host for a period of time, reducing the potential for that host to compromise an asset. The identification of which of your hosts are most frequently identified in alerts can help you to spot your weakest, most vulnerable, or at least most targeted systems. These systems likely warrant additional scrutiny.

Now that we have some good questions to start with, we have to figure out how to go about answering them.

Command Line Kung Fu

- If you answered, "With Grep, Cut, Sort, and Uniq!" you're right!
 - Let's take a few minutes together for your instructor to walk you through a few examples.
 - It would be wise to ask questions and take notes!

If you hear those questions and immediately think, "Command line!" then you have learned well the lessons we have sought to teach you. Yes, indeed, slicing and dicing the alert log with a combination of careful greps, awks, cuts, and sorts is just what we need to do. Rather than simply give you printed answers here in the book, we want you to work this out; frequently, we remember things that we have to figure out ourselves.

If you are taking the class live, your instructor will have a discussion at this point and work with your class to figure out how to answer these three questions using the command line. If you are taking the class OnDemand, a recording has been prepared that you can view, working through these questions one by one.

Whichever way you are taking the class, we recommend that you really think about things as the instructor is working them out. You will be trying to apply these same techniques to solve your bootcamp problems!

Section 1: Introduction	Section 2: Fundamentals	Section 3: Bootcamp	Section 4: Advanced Topics	Section 5: Real-World Application
- An immersion class in speaking packets, protocols, and hex - Ethernet - IPv4 & IPv6 - TCP, UDP, & ICMP - Wireshark & tcpdump fundamentals, filtering, & investigation	- Scapy for building tools and simulating signatures & behaviors - Snort & Suricata - Researching common application protocols	- Architecting for massive packet capture & analysis - Overcoming the limitations of signature-based tools - Basic Zeek usage and logs - Threat modelling - Advanced correlation using Zeek	- Guided scenarios to reinforce all skills - Elaboration on effective deployment, collection, and analysis at large scale - Moving beyond alerts to data, driven analysis, & machine learning	Challenge questions guide/follow expert analysts' progression through real-world data

Bootcamp (Sections 1-5): Real-World Application of Theory

• Bootcamp

Section 3 Bootcamp

Processing Snort Alerts

Researching Protocols

Researching SMB

Cleartext Protocols

There are many cleartext protocols:

- Telnet, SMTP, IMAP, POP, FTP, Syslog, etc.

Generally, very easy to parse and understand

- Most will have plaintext "headers"
- Similar to looking at HTTP, but different headers, obviously

We won't take time to take these apart.

- Usually, you can figure them out intuitively.
- If you need help, read the RFC!
- These days, most have TLS hooks to switch over the encrypted communications.

When it comes to researching protocols, some are very easy to figure out. For example, there are still a large number of plaintext protocols. When we say this, we are implying something more than, "They're not encrypted." For example, NTP isn't encrypted, but we don't normally put it into this same grouping.

What do we mean, then? Plaintext protocols are not only unencrypted (at least, they might be unencrypted), but the protocol headers are in plaintext as opposed to being a binary header. DNS, for example, uses a binary header. Even though DNS over port 53 isn't encrypted, we don't usually describe it as a plaintext protocol since you need to know a bit about the header structure to figure out what's happening.

Protocols like telnet are not only unencrypted, but have very minimal, if any, headers. (Telnet actually does send some control codes over the channel, but overall, it is header-free.). FTP is also plaintext in that it isn't traditionally encrypted and in that the headers themselves are very easy to read, parse, and understand since there is no binary encoding used. The same is true of protocols like IMAP, POP, Syslog, and others.

When you are facing a protocol like this, you can usually figure out what's happening without ever looking at the RFC. If you run into a situation where there's something unusual happening and you need to be sure what a header says, that's the time to head to the RFC. You may find that some binary protocol headers are also very straightforward. It is not unusual for someone who has a good facility with reading hex and a deep understanding of network protocols generally to be able to figure out, or reverse engineer, a binary protocol header without having any documentation available.

One obstacle to this, however, would be the shift that has occurred to using TLS to encrypt most protocols. SMTP, IMAP, and other plaintext protocols have support to opportunistically switch to TLS using a TLS handshake. Can you still inspect these? Maybe. We'll address that more clearly in Book 4.

Identifying Connections

- In a situation where you are attempting to understand a network or general activity on a network, it makes sense to figure out which protocols are in use:
 - How can you identify all of the TCP protocols in use?
 - How can you isolate a particular protocol to analyze it further?
 - How can you research a protocol that you haven't seen before?
- Answering these questions will be the focus of our discussion
 - You should already know the answer to the third question... It's what you've been doing for the last two sections!

There are situations where an analyst will know exactly what he or she is looking for. This could be because of the way that an alert was raised, the specific request made by a security team or administrator, or deep knowledge of the network in question. Quite often, however, the analyst will not know exactly what is interesting. In fact, even in situations where we *think* that we know what matters, there is a great deal of value in standing back and looking at the data in the context of all of the rest of the network behavior. This allows us to objectively determine what previous activity led up to the activity of interest. This allows us to form and subsequently prove or disprove hypotheses about the activity.

To do this well, we must be able to answer several questions effectively: Which protocols are in use on the network? Of the protocols that are in use, is it possible to isolate and extract that data for further analysis? When analyzing the extracted data, is it possible to determine precisely what the traffic is and what its purpose is? For this bootcamp, we are focusing on how to go about answering these questions through an easy-to-follow process.

How to Identify Protocols in Use

- Generally, protocols in use are equal to service ports in use
 - In other words, if someone establishes a TCP connection to port 389, port 389 is running some kind of application layer protocol that the client wishes to consume
- What strategy would you use to identify all of the service ports that are in use?
 - Could BPF help here?

To answer the first question, we want to identify all of the protocols that are in use on the network during the period of time in question. At a network level, this would mean IP versus IPv6 or other non-IP level protocols. We could even look further down and see if there are other link layer encapsulations in use. Moving up the stack, we would be interested in any protocols other than ICMP, TCP, and UDP. Finding things of this sort is not difficult. For example:

```
tcpdump -r capture.cap -n 'not(ip[9]=6 or ip[9]=17 or ip[9]=1)'
```

This would quickly show us if we had any transport layer protocols in use other than TCP, UDP, and ICMP. We can do similar things to identify different link layers and different network layers in use. Can we also identify application layer protocols? Absolutely.

Doing so is most easily done using BPF to identify connections or connection attempts. Once this is done, we could extract the related data and begin analysis.

Using BPF to Identify Protocols/Services/Ports

- We can search for all of the unique destination ports where SYN is set alone
 - These are connection attempts: `tcp[13]&0x3f=0x02`
 - A connection attempt does not mean the service is listening!
- We can also search for all of the unique source ports where SYN/ACK is set!
 - SYN/ACK means that the service is listening!
 - `tcp[13]&0x3f=0x12`
- Put this together with some command-line Kung Fu!

```
tcpdump -r backbone.cap -n 'tcp[13]&0x3f=0x02' | cut -d ' ' -f 5 |
cut -d . -f 5 | sort | uniq -c | sort -nr > conn_attempts
```

There are two basic approaches to finding TCP services: We can either search for connection attempts or search for responding services. There is merit to both approaches.

If we search for connection attempts, we may find SYN packets sent to ports where there is no service listening. While these can certainly be distracting, if we are looking at a capture taken from an internal network, it is still very interesting. Even if the TCP port to which the SYN was sent is closed, we must ask why that SYN was sent. Is the client host misconfigured? Was it a simple error? Is there evidence that a scan of some type is being run? If our analysis reveals that it was a scan, we must now ask whether or not the scan was authorized and/or expected. If it wasn't, we may have just found something that is an indicator of compromise!

If, on the other hand, we search for SYN/ACK packets, indicating that a TCP port is listening, we will not find scans to closed ports; we will, however, be sure that every port that we find is actually listening and a connection was successful! This can narrow things down to ports where data has actually transferred.

If we take the former approach, we will want to focus our attention on unique destination ports. If we take the latter approach, we will want to focus on unique source ports. In the slide, we show an example of using the former approach to extract all of the unique destination ports to which a SYN was sent, as viewed from our backbone sensor. The unique ports, along with a count of how many times each was attempted, is sent to a file named 'conn_attempts'.

How to Isolate a Protocol for Analysis

- Now that we have some ports that are interesting, how can we extract the data?
 - tcpdump: Our old friend! Specify '-r' and '-w', along with a BPF filter
 - tshark: Yes, this works, too; '-r' and '-w', but with the benefit of display filters; SLOW!
 - editcap: Read, write, edit, slice, etc. Can extract packets based on time, filter, count, etc.
- What might this look like?

```
tcpdump -r backbone.cap -n -w 88.cap 'tcp port 88'
```

Now that we have a list of all of the ports involved in interesting activity, how can we isolate those connections for deeper analysis? Three possible options are tcpdump, tshark, and editcap.

Thus far, we have used tcpdump to read and display packets. It is possible to both read from a packet capture file and write to a new file at the same time. Doing this really doesn't make sense unless you are also specifying some type of BPF filter to reduce the data in the input to a subset in the output. For example, imagine that we found that there were a large number of connections to TCP port 88 across our backbone. Wishing to research this further, we can extract the port 88 traffic using a BPF filter. An example of this is shown in the slide, writing all of the TCP port 88 packets into a file named '88.cap'. Once this has completed, we can look at that file with a tool like Wireshark.

The tshark tool can also be used for this. Be aware that using tshark can have a huge performance impact. For example, running the command in the slide against the backbone.cap data file provided in the /final directory takes just under 30 seconds to complete. Performing the same action using tshark takes nearly two minutes! That might not seem like much, but the larger the packet and the more complex the filter, the more this will be magnified!

The final option that we will present is editcap. This is an extremely powerful tool for manipulating large packet captures. It includes options that allow you to slice a large packet capture into small, manageable pieces. It allows you to extract data based on a BPF filter, much as we have with tcpdump in this example. One of the most powerful features, however, is the ability to extract data based on a starting and ending time.

Why is this useful? Imagine that you have identified a machine involved in what appears to be odd behavior. You are interested in finding out what else was happening on the network immediately before and after the strange activity. You can't define a BPF filter for this because you don't know which hosts are involved and what they are doing. Using editcap, however, you can specify a starting time a few seconds before the activity and an ending time a few seconds later, allowing you to extract *all* of the packets around that time!

Researching a Protocol

- Now that you have the data, you can begin researching.
- Suggested steps:
 - Before investing a great deal of effort, look at the behavior of the hosts.
 - For example, if none of the connections complete, there's little sense in going further.
 - Still, think about why it's happening. Is it a scan? Is it a misconfiguration? What's your opinion?
 - If connections complete, use Wireshark or tcpdump to view some connections.
 - Does Wireshark seem to know what it is? Do the decodes make sense?
 - What can you find when you google for "TCP" and the port number?
 - Are there interesting results if you add some plaintext packet content to your search?
 - Examine all of this in context to try to determine what the purpose of the traffic is.

Once you have successfully extracted data that is potentially interesting, it's time to analyze it. As a suggestion, take a high-level view of the data first. Is there evidence that the connections complete? If not, could the data be evidence of a scan of some kind? If so, can you identify who the scanner or scanners are? If it doesn't appear to be a scan, is it possible that you are looking at a misconfigured host?

If the data reveals that there is more than a simple scan or misconfiguration, the next step would be to take an excerpt of that data and look at it in a tool like Wireshark. Does Wireshark seem to know what it is? Even if Wireshark *claims* to know which protocol it is looking at, look closely at the decodes; do they seem to make some sense, or are they full of junk data? Wireshark can get things wrong, especially when the protocol dissector is keyed only to the port number. Based on what you see in Wireshark, what can you find in Google to better understand the behavior? If Wireshark doesn't have a dissector, consider taking plaintext portions of the packet capture and searching for those in Google, along with the TCP port number.

Based on all this research, can you come up with an explanation of what the packets are for and what the hosts are attempting to do? When you can, it's time to move to the next protocol!

Section 1: Fundamentals	Section 2: Tools & Techniques	Section 3: Real-World Application	Section 4: Advanced Topics	Section 5: Capstone Project
- An immersion class in speaking packets, protocols, and hex - Ethernet - IPv4 & IPv6 - TCP, UDP, & ICMP - Wireshark & tcpdump fundamentals, filtering, & investigation	- Scapy for building tools and simulating signatures & behaviors - Snort & Suricata - Researching common application protocols	- Architecting for massive packet capture & analysis - Overcoming the limitations of signature-based tools - Basic Zeek usage and logs - Threat modelling - Advanced correlation using Zeek	- Guided scenarios to reinforce all skills - Elaboration on effective deployment, collection, and analysis at large scale - Moving beyond alerts to data, driven analysis, & machine learning	Challenge questions guide/follow expert analysts' progression through real-world data

Bootcamp (Sections 1-5): Real-World Application of Theory

• Bootcamp

Section 3 Bootcamp

Processing Snort Alerts

Researching Protocols

Researching SMB

Microsoft Protocol General Detection Challenge

Microsoft protocols are poorly documented and incredibly complex,

- It's not so much that Microsoft *tried* to make them hard; it's that they have so many developers adding and working on so many features and they all decide to do things differently!
 - For example, SMB1 uses two separate 16-bit fields that must be aggregated to calculate 32-bit values in the protocol header rather than a single 32-bit field.

The Network Gods above have said: *Thou Shalt Use Network Byte Order!*

- Many years ago, NBO was established as *the standard* for network protocols.
 - NBO = big endian
- Parts of Microsoft's protocols are big endian, parts are little endian, parts use ASCII, parts use UNICODE...

The Microsoft suite of protocols is ubiquitous, complex, and very poorly documented. This creates some big problems for analysis and big challenges when it comes to writing rules and other detection capabilities. In our opinion, Microsoft likely did not set out to create a complex protocol. Instead, we believe that the complexity is an artifact of a few circumstances. One is that Microsoft has long been very committed to ensuring backward compatibility with *everything*. In the last few years, there has been some change with this, but it is actually still possible to get a Windows NT (1990) computer to access a file share on a Server 2019 server. While that's very impressive, it means that a great deal of dross has collected in the protocol over the years.

Another circumstance that works against Microsoft is that it has likely had hundreds, if not thousands, of internal developers working on, adding to, modifying, and otherwise touching the protocol. Each of those programmers has brought his or her own beliefs, perspectives, and biases to the modifications made to the code and the protocol. Since the only public documentation that Microsoft has ever released for the protocol was released as a result of a court *forcing* it to release documentation, you can imagine that the official documentation is scant and inaccurate (for more information, see https://www.theregister.co.uk/2007/12/21/samba_microsoft_agreement/).

One of the things that sets Microsoft's protocol apart from other modern network protocols is that it does not obey some of the "rules of the road." Many, many years ago, the idea of "Network Byte Order" was established. This term refers to the ordering of bytes when representing integers in network protocol headers. Network Byte Order was defined to always be big endian (if you're not sure what this means, refer to this Wikipedia article: <https://en.wikipedia.org/wiki/Endianness>). Host Byte Order, by contrast, is *whatever endianness is native to the architecture of your system*. In other words, it's not actually defined.

Unfortunately, not only does the Microsoft protocol use ASCII in some places and UNICODE in others, but it also uses Network Byte Order (big endian) in some places and little endian in others! This is an analysis nightmare.

Server Message Block

Server Message Block allows sharing of files, directories, printers, and other network resources.

- Microsoft's adoption and extension of CIFS
- It's not CIFS... It's just based on CIFS.

Facilities for finding and identifying shared resources

Facilities for authenticating and authorizing access to resources

Supports interprocess communications

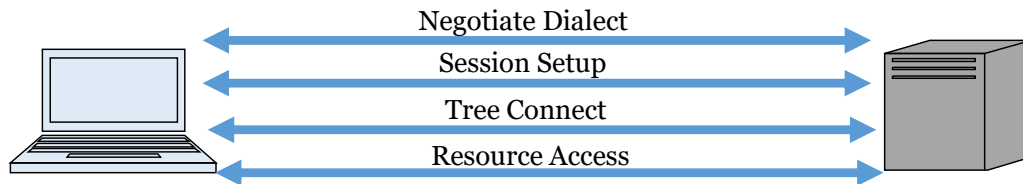
SMB is used to share network resources such as files and printers. It is also an intricate part of any Windows network and can be difficult to understand when examining all the many calls and transactions made to access network resources. SMB can be used as a transport for MSRPC, so it is quite useful to understand what SMB does and how it interacts with MSRPC.

Originally, Microsoft adopted the CIFS (Common Internet File System), which was a file sharing protocol developed at IBM in the 1980s. When Microsoft adopts a protocol, however, it always "extends" the protocol (makes it incompatible with proprietary extensions), so SMB really is not CIFS and hasn't been for many years.

SMB is a commonly seen Windows protocol used for network resource sharing. SMB is a client-server application where the client makes individual requests, such as opening a file, and the server responds to the specific request. File sharing is at the heart of SMB, but it also provides functionality for *interprocess communication (IPC)*. SMB implements remote file access in a manner that allows many applications to share data on local disks and file servers.

SMB supports access to files, shared printers, an interprocess communication *named pipe*, and serial or other communications devices. SMB must provide functionality for finding and identifying these shared resources. It must also include some means of authenticating the user and authorizing access to the requested resource.

SMB Conversation Flow



Potential Issues:

- SMB fragments
- Out-of-order SMB fragments
- Compounding SMB commands using AndX
- Padding after SMB heading and/or between SMB commands
- Out-of-order SMB AndX messages

There is some good news. Pretty much every Microsoft connection that you ever look at will follow a standard conversation flow, going through four main phases. The first transaction is a negotiation of the SMB *dialect*. Over the years, different versions of Windows have used different SMB functions and calls. An SMB dialect indicates which SMB functions are available. During the initial SMB connection, the client and server must negotiate the dialect to be used; the client announces its supported dialects, and the server selects the most appropriate. There are many different versions of SMB, such as Samba, SMB 2.x., and SMB 3, requiring the two communicating hosts to agree upon a particular dialect that they both support.

Next, the session setup request is required to authenticate the user for the requested resources. The exact type of authentication used depends on the operating system and the network configuration. If the Windows host is part of an Active Directory environment, it may use Kerberos authentication. If Active Directory is not supported, a *Security Accounts Manager (SAM)* database may be used to store and retrieve user IDs and passwords.

The SMB *tree connect* provides access to the requested share. Next, the requested resource is opened for processing, including reading, writing, or overwriting an existing file. Finally, access to the file should be closed, as should access to the share via a *tree disconnect*. The *AndX* allows several SMB transactions to be piggybacked and included in a single packet for the sake of efficiency.

Opportunistic SMB evasions can occur when the IDS has no way of reassembling SMB fragments. As explained, this is different from IP fragmentation; the fragmentation occurs in the SMB portion of the data only. SMB fragments must be reassembled and reordered to properly examine content. The IDS/IPS must be aware of and capable of examining the multiple SMB commands that may be present when found in AndX messages. Also, padding can be used in the SMB header or data to align on a memory boundary. The IDS/IPS must be capable of recognizing and discarding meaningless pad bytes. Finally, AndX messages may be generated that do not have the expected sequence of processing, and the IDS/IPS must reorder them in chronological progression.

SMB Session to Browse a Share

No.	Source	Destination	Protocol	Src Port	Dst Port	Info
1	192.168.60.22	192.168.50.30	TCP	49799	445	49799 → 445 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256
2	192.168.50.30	192.168.60.22	TCP	445	49799	445 → 49799 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=
3	192.168.60.22	192.168.50.30	TCP	49799	445	49799 → 445 [ACK] Seq=1 Ack=1 Win=65536 Len=0
4	192.168.60.22	192.168.50.30	SMB2	49799	445	445 Negotiate Protocol Request
5	192.168.50.30	192.168.60.22	SMB2	445	49799	445 Negotiate Protocol Response
6	192.168.60.22	192.168.50.30	SMB2	49799	445	445 Session Setup Request
7	192.168.50.30	192.168.60.22	TCP	445	49799	445 → 49799 [ACK] Seq=253 Ack=1879 Win=65536 Len=0
8	192.168.50.30	192.168.60.22	SMB2	445	49799	445 Session Setup Response
9	192.168.60.22	192.168.50.30	SMB2	49799	445	445 Tree Connect Request Tree: \\fileserver\IPC\$
10	192.168.50.30	192.168.60.22	SMB2	445	49799	445 Tree Connect Response
11	192.168.60.22	192.168.50.30	SMB2	49799	445	445 Ioctl Request FSCTL_DFS_GET_REFERRALS, File: \fileserv
12	192.168.60.22	192.168.50.30	SMB2	49799	445	445 Ioctl Request FSCTL_DFS_GET_REFERRALS, File: \fileserv
13	192.168.60.22	192.168.50.30	SMB2	49799	445	445 Ioctl Request FSCTL_DFS_GET_REFERRALS, File: \fileserv
14	192.168.50.30	192.168.60.22	TCP	445	49799	445 → 49799 [ACK] Seq=598 Ack=2327 Win=65024 Len=0
15	192.168.60.22	192.168.50.30	SMB2	49799	445	445 Ioctl Request FSCTL_DFS_GET_REFERRALS, File: \fileserv
16	192.168.50.30	192.168.60.22	TCP	445	49799	445 → 49799 [ACK] Seq=598 Ack=2651 Win=64768 Len=0
17	192.168.60.22	192.168.50.30	SMB2	49799	445	445 Ioctl Request FSCTL_DFS_GET_REFERRALS, File: \fileserv
18	192.168.50.30	192.168.60.22	SMB2	445	49799	445 Ioctl Response, Error: STATUS_NOT_FOUND
19	192.168.50.30	192.168.60.22	SMB2	445	49799	445 Ioctl Response, Error: STATUS_NOT_FOUND
20	192.168.60.22	192.168.50.30	TCP	49799	445	445 → 49799 [ACK] Seq=2823 Ack=752 Win=64768 Len=0
21	192.168.60.22	192.168.50.30	SMB2	49799	445	445 Tree Connect Request Tree: \\fileserver\admin
22	192.168.60.22	192.168.50.30	SMB2	49799	445	445 Tree Connect Request Tree: \\fileserver\accounting
23	192.168.50.30	192.168.60.22	TCP	445	49799	445 → 49799 [ACK] Seq=752 Ack=3057 Win=64512 Len=0
24	192.168.50.30	192.168.60.22	SMB2	445	49799	445 Tree

See the phases? Negotiate Protocol, Session Setup, Tree Connect, Resource Access

SANS



smb.pcap

SEC503: Intrusion Detection In-Depth 153

Pictured here, we have an example of a session where a user is accessing a file share or a series of files shares, depending on how you look at it (all the shares being accessed are on the same server, so it is arguable whether these are different "shares" or just different "directories"). Try not to focus on the details of the packets just yet. We know that you've become accustomed to diving right into the hex, but we're going to stand back a bit as we look at Microsoft protocols and allow our tool (Wireshark) to do the work for us.

If you peruse the "Info" column, you should be able to pick out the major phases of SMB activity. Notice that after the three-way handshake, we see the SMB protocol negotiation occurring in packets 4 and 5. Next, a session setup occurs in packets 6 and 8. This is where authentication activities would occur, if required. With that completed, we move to a tree connect, which is where we begin to be able to identify the resource or resources being accessed. In this case, we see a connection to the IPC\$ share (interprocess communication, though Microsoft seems to apply this to any kind of communication). Following that, we see a resource access occurring in the remainder of the packets until we see additional connection requests in packets 21 and 22, where individual shares or directories are being accessed.

Again, standing back and looking for the phases is one of the easiest ways to understand what's happening in Microsoft communications. Also, bear in mind that you may see the connection switch between tree connections and resource access multiple times in a single, long session, so don't stop looking when you find the first tree connect!



You can view these packets using:

wireshark smb.pcap

Wireshark to the Rescue!

Wireshark's protocol dissectors for Microsoft protocols are great!

- Easily add columns to your display for things that actually matter
- Easily track activity from a certain host or on a server
- Easily identify files and other resources being accessed

Zeek also has a very functional SMB protocol analyzer:

- Supports SMB1 and SMB2
 - While there are many versions after SMB 2.0, they are effectively just modifications to 2.
 - The SMB2 events fire just fine for SMB3 transactions.

As we have already begun to see, Wireshark is a real lifesaver when trying to analyze an incident involving Microsoft protocols. We will explore some of the other awesome things we can do to make our lives easier in a few moments. This includes things like adding and customizing columns to our display, making it a breeze to track activity in an SMB session.

We will discuss the Zeek (formerly Bro) tool in Book 4, but it is worth mentioning now that Zeek has very complete support for SMB 1 and 2 as well. This will allow you to generate logs regarding a great deal of interesting SMB activity without needing to dig into packets, which is great for continuous monitoring.

You might wonder how useful it is to support only SMB 1 and 2. After all, Microsoft is now pushing SMB 3 in Windows 2016 and Windows 10. There's some good news here.

While there are some minor changes in SMB 3 and SMB 3.1.1, the protocol is actually largely unchanged from SMB 2. The biggest difference is the support for native encryption of the data layer (which is not mandatory, by the way). This means that, at least for now, both Wireshark and Zeek are wonderful tools for dealing with Microsoft protocols.

SMB2/3 Message Types

19 different messages; only 10 are very common

- | | |
|-----------------------------------|-----------------------|
| • <u>NegotiateProtocol</u> | • <u>Write</u> |
| • <u>SessionSetup</u> | • Lock |
| • <u>SessionLogoff</u> | • <u>IOCtl</u> |
| • <u>TreeConnect</u> | • Cancel |
| • <u>TreeDisconnect</u> | • KeepAlive |
| • <u>Create</u> | • Find |
| • <u>Close</u> | • Notify |
| • Flush | • GetInfo / SetInfo |
| • <u>Read</u> | • Break |

Even though there are 19 different SMB message types, there are only 10 that you will usually see in practice:

- **NegotiateProtocol** is used exclusively to determine which dialect of SMB the two hosts have in common.
- **SessionSetup** is used for user authentication negotiation. This will establish the credentials used for every other activity in this session.
- **SessionLogoff** is seen only at the very end of the session, immediately before the session closes. You can think of it as a "logoff" request.
- **TreeConnect** is used to create a connection to a resource of some kind. This could be (and is usually) the IPC\$ share, though it might be to a file share, a named pipe, or some other service on the server.
- **TreeDisconnect** is a notification to the server that the connection to the associated resource is being released. You could think of it as unmounting a share, though it applies to named pipes and other resources, not only file shares.
- **Create** sounds as though it will create a file, but typically this is executed as a "Create or Open." It might be better to think of this as "Create a resource handle to let me access something" rather than as "Create a file."
- **Close** is analogous to the TreeDisconnect in that it releases the file or other resource that was being accessed.
- **Read** will, not surprisingly, read data from a resource.
- **Write** will, as expected, write data to a resource.
- **IOCtl** executes a file system or device control call across the network. This would be something along the lines of peeking into a file or device, sending a copy request asking for a file to be copied from one directory to another on a file server, reading file attribute information, etc.

Wireshark: Associating Requests/Responses

No.	Time	Source	Destination	Protocol	Src Port	Dst Port	Info
4	0.0...	192.168.60.22	192.168.50.30	SMB2	49799	445	Negotiate Protocol Request
5	0.0...	192.168.50.30	192.168.60.22	SMB2	445	49799	Negotiate Protocol Response
6	0.0...	192.168.60.22	192.168.50.30	SMB2	49799	445	Session Setup Request
8	0.0...	192.168.50.30	192.168.60.22	SMB2	445	49799	Session Setup Response
9	0.0...	192.168.60.22	192.168.50.30	SMB2	49799	445	Tree Connect Request Tree: \\fileserver\IPC\$
10	0.0...	192.168.50.30	192.168.60.22	SMB2	445	49799	Tree Connect Response
11	0.0...	192.168.60.22	192.168.50.30	SMB2	49799	445	Ioctl Request FSCTL_DFS_GET_REFERRALS, File: \fil
12	0.0...	192.168.60.22	192.168.50.30	SMB2	49799	445	Ioctl Request FSCTL_DFS_GET_REFERRALS, File: \fil
13	0.0...	192.168.60.22	192.168.50.30	SMB2	49799	445	Ioctl Request FSCTL_DFS_GET_REFERRALS, File: \fil

▶ Frame 6: 1824 bytes on wire (14592 bits), 1824 bytes captured (14592 bits) on interface 0
 ▶ Ethernet II, Src: 00:50:56:86:27:f6, Dst: 00:50:56:86:d4:c3
 ▶ Internet Protocol Version 4, Src: 192.168.60.22, Dst: 192.168.50.30
 ▶ **Transmission Control Protocol, Src Port: 49799, Dst Port: 445, Seq: 109, Ack: 253, Len: 1770**
 ▶ NetBIOS Session Service
 ▼ SMB2 (Server Message Block Protocol version 2)
 ▼ SMB2 Header
 Server Component: SMB2
 Header Length: 64
 Credit Charge: 1
 Channel Sequence: 0
 Reserved: 0000
 Command: Session Setup (1)
 Credits requested: 31
 ▶ Flags: 0x00000000
 Chain Offset: 0x00000000
 Message ID: Unknown (1)

All SMB messages except the Negotiation have a Message ID. Track them to track activity!



A wonderful key that you can use to understand what is happening in your SMB traffic is the **Message ID** field. You will find that this field exists in every SMB request and response, though it is typically empty (set to zero) in the protocol negotiation request and response.

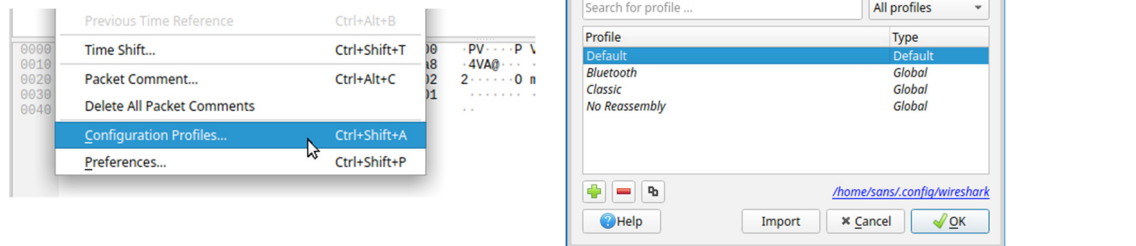
The value in this field, which you can see highlighted in the slide in the packet details, increments for every request within a given SMB session. This means that you can tell in which order requests were sent and even processed.

While these numbers will always be sent sequentially, you might find that they do not always come back sequentially. It is possible that a response is delayed and that another request is handled or completed first. Still, these numbers allow you to easily discover how a particular request turned out.

Aside: Wireshark Profiles

We're about to start changing around Wireshark.

- Before we do, we want to talk to you about "Profiles."
- Click on "Profile" in the lower right to see what you already have.
- Use "Configuration Profiles" to manage them.



In just a moment, we're going to start changing around how Wireshark displays things. While this is totally fine, keep in mind that Wireshark will remember whatever you do to it. This means that if we add in or delete a bunch of columns from the display, exactly the same view will be presented to us when we restart Wireshark. While this is cool and useful, it can also be disconcerting if you aren't ready for it to happen.

To see what profiles are already available, you can click on the "Profile" text in the lower right corner of the Wireshark window. From here, you can choose any already defined profile, but you cannot create a new one.

To create a new profile, pull down the "Edit" menu and select "Configuration Profiles." This will open the dialog box pictured in the lower right of the slide. From here, you can duplicate an existing profile, remove a profile, or add a profile. If you add a profile, it will typically create a fresh copy of the Default profile and allow you to assign a new name to it.

Once you have created a profile and named it, you can close this dialog. Verify that the lower right corner shows the name of the profile that you just created, and then make whichever changes you want to Wireshark's display!

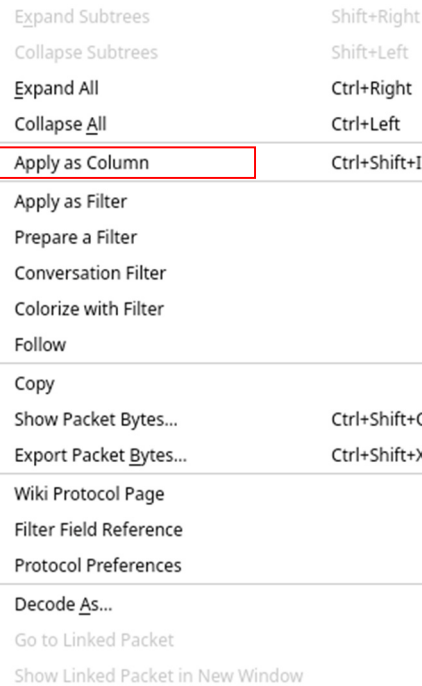
Wireshark: Add Columns

Apply as Column

- Right-click on anything interesting.
- For us, use the "Message ID" in the header.

Automatically adds a column to your Wireshark view!

Take the opportunity to try this in the lab!



Let's go back to our SMB discussion. We saw the Message ID and said that it can be used to figure out which request goes with which response. In fact, if you right-click on the Message ID field in the packet details, you can choose to use the "**Apply as Column**" option. This will instantly add that field to the Wireshark columns in your display. If you have created a new profile, this column is added to the profile that you currently have selected. This means we could fully customize how data is displayed based on a specific profile that might be tied to a particular protocol that we have on our network! When we want to go back to the normal view, we can just select Default again. Before we move on, we should mention that you can also use the + button to the right of the display filter input field to add "Buttons" to your Wireshark display. This allows you to create a button that will automatically apply any filter that you define to your view. This can make Wireshark a very powerful and very efficient incident analysis tool.

For now, we will look at adding some useful columns related to SMB, but remember that this feature is available. If you are analyzing any kind of data and find something useful in the packet details that you would like in your packet list, simply right-click on it and add it!

This might seem just a little bit familiar. Have we seen this kind of capability before? Yes and no. We haven't seen it directly in Wireshark, but remember that when we looked at tshark, we discovered that not only could we use display filters on the command line, but we could select specific fields from the packet headers that we wanted displayed. This is really that same feature but in the GUI.

Wireshark: Correlating Requests/Responses

No.	Source	Destination	Protocol	Src Port	Dst Port	Message ID	Info
4	192.168.60.22	192.168.50.30	SMB2	49799	445	0	Negotiate Protocol Request
5	192.168.50.30	192.168.60.22	SMB2	445	49799	0	Negotiate Protocol Response
6	192.168.60.22	192.168.50.30	SMB2	49799	445	1	Session Setup Request
8	192.168.50.30	192.168.60.22	SMB2	445	49799	1	Session Setup Response
9	192.168.60.22	192.168.50.30	SMB2	49799	445	2	Tree Connect Request Tree: \\fileserver\IPC\$
10	192.168.50.30	192.168.60.22	SMB2	445	49799	2	Tree Connect Response
11	192.168.60.22	192.168.50.30	SMB2	49799	445	3	Ioctl Request FSCTL_DFS_GET_REFERRALS, File: \f
12	192.168.60.22	192.168.50.30	SMB2	49799	445	4	Ioctl Request FSCTL_DFS_GET_REFERRALS, File: \f
13	192.168.60.22	192.168.50.30	SMB2	49799	445	5	Ioctl Request FSCTL_DFS_GET_REFERRALS, File: \f
15	192.168.60.22	192.168.50.30	SMB2	49799	445	6	Ioctl Request FSCTL_DFS_GET_REFERRALS, File: \f
17	192.168.60.22	192.168.50.30	SMB2	49799	445	7	Ioctl Request FSCTL_DFS_GET_REFERRALS, File: \f
18	192.168.50.30	192.168.60.22	SMB2	445	49799	6	Ioctl Response, Error: STATUS_NOT_FOUND
19	192.168.50.30	192.168.60.22	SMB2	445	49799	7	Ioctl Response, Error: STATUS_NOT_FOUND
21	192.168.60.22	192.168.50.30	SMB2	49799	445	8	Tree Connect Request Tree: \\fileserver\admin
22	192.168.60.22	192.168.50.30	SMB2	49799	445	9	Tree Connect Request Tree: \\fileserver\account
24	192.168.50.30	192.168.60.22	SMB2	445	49799	9	Tree Connect Response
25	192.168.50.30	192.168.60.22	SMB2	445	49799	8	Tree Connect Response
27	192.168.50.30	192.168.60.22	SMB2	445	49799	3	Ioctl Response, Error: STATUS_NOT_FOUND
28	192.168.50.30	192.168.60.22	SMB2	445	49799	4	Ioctl Response, Error: STATUS_NOT_FOUND
29	192.168.60.22	192.168.50.30	SMB2	49799	445	10	Tree Connect Request Tree: \\fileserver\common

Correlate requests and responses; sometimes there are multiple responses in a single packet.



Now that we have added the Message ID column, you can see how simple it is to identify which response goes with which request. You can also see, if you look at packets 12 through 28, that Message ID 4 is sent in packet 12, but the response doesn't return until packet 28. I think you can imagine trying to figure out how packet 13 is related to packet 12... and it turns out that they aren't really related, except that they are requests in the same session. Knowing how the ID can be correlated, however, can really make things easier.

Don't forget that you can sort the columns in Wireshark. In other words, if you wanted to, you could click on the Message ID column to sort all the packets in ascending or descending order based on the message ID. If you do choose to do this, remember that *Wireshark will no longer care which session they are a part of!* Therefore, if you choose to sort by message ID, we would recommend first filtering down to just one session so that you aren't inadvertently trying to correlate messages from different sessions.

A More Interesting View

No.	Source	Destination	Message ID	NT Status	Tree	Filename
39	192.168.50.30	192.168.60.22	13	STATUS_OBJECT_NAM...	\\fileserver\admin	
40	192.168.60.22	192.168.50.30	14		\\fileserver\admin	
41	192.168.50.30	192.168.60.22	14	STATUS_SUCCESS	\\fileserver\admin	
42	192.168.60.22	192.168.50.30	15		\\fileserver\admin	
43	192.168.60.22	192.168.50.30	16,17		\\fileserver\admin,\\file...	
45	192.168.50.30	192.168.60.22	16,17	STATUS_SUCCESS, ST...	\\fileserver\admin,\\file...	...,Accounts Passwords
47	192.168.50.30	192.168.60.22	15	STATUS_PENDING		
48	192.168.60.22	192.168.50.30	18		\\fileserver\admin	

- FileIdBothDirectoryInfo: ..
- FileIdBothDirectoryInfo: Accounts Passwords.txt
- FileIdBothDirectoryInfo: FTP LOGIN PASSWORD SHEET.txt
- FileIdBothDirectoryInfo: IFDS Passwords.txt
- FileIdBothDirectoryInfo: Logins and Passwords.txt
- FileIdBothDirectoryInfo: Master Application List.txt
- FileIdBothDirectoryInfo: My Passwords.txt
- FileIdBothDirectoryInfo: NOVEMBER 2012 NETWORK A R REPORT PASSWORD.xls
- FileIdBothDirectoryInfo: Password Admin account chart.txt
- FileIdBothDirectoryInfo: Password Reset - Email.xlsx

You can add any columns you'd like to, and don't forget to look at the detail view too.



As you can see, we have added a few more fields to our view, in addition to removing some columns that weren't particularly useful when looking at SMB traffic. We'll look at how to manage these columns more efficiently in a few minutes.

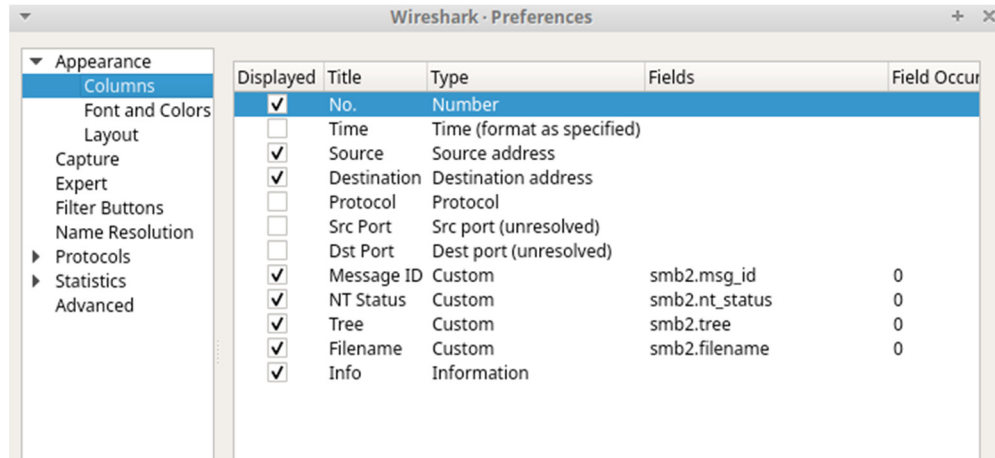
Looking at our view now, we not only can see the message IDs, but we can see the **NT Status** field, which is the result code for the request. While we're not currently filtering on any of these, I think you can imagine that you could locate something like an "ACCESS_DENIED" and then perhaps add that as a filter, allowing you to find every failed access attempt in a session! This can make things very easy!

We've also added the **Tree** as a column to more easily see what's being accessed. An additional **Filename** column has been added as well. If you take note of packet 45, which is highlighted, you can see that this actually has the responses to two separate requests in it and more than one filename. In fact, when adding all of these useful columns, don't forget to still look into the packet details. In the packet details, we can see a number of very interesting sounding filenames.

It's worth mentioning that, as a result of the complexity of SMB, you will find that there are multiple fields named "Filename" in various SMB messages. Therefore, if you are trying to look at file share access versus some other kind of file access, it is worth looking at multiple message types to verify that you have located all the fields of interest.

Save Useful Column Combinations

- You can change/save your column combinations through the Preferences settings.



After adding all of these useful fields and removing or turning off the fields that didn't matter for SMB, you will find that if you exit out of and then restart Wireshark, your columns are still set up for SMB analysis. This is easily remedied.

Using the **Edit** menu, pull up the **Preferences** pane. Within that panel, under **Appearance**, you can choose the **Columns** option, which is pictured in the slide. Here you can see all the currently configured columns, whether they are displayed or not. Rather than removing all the hard-won fields that we've added in, it makes far more sense to simply uncheck them while simultaneously turning the normal fields back on.

Remember that these preferences are saved in a Wireshark configuration file, so if you have useful column setups, you can share these not only with your other workstations but also with your colleagues.

Microsoft RPC Implementations

RPC over SMB:

- Uses SMB session to connect to RPC services
- Performing authentication and endpoint mapping via SMB

RPC directly over TCP/UDP (also known as DCOM):

- Uses RPC Endpoint Mapper (TCP/UDP port 135) to find/connect to requested RPC service port
- Uses RPC to connect to discovered port

RPC over HTTP or HTTPS:

- Tunnels RPC requests over well-known ports
- Uses HTTP RPC Endpoint Mapper (TCP/UDP port 593) to find requested RPC service port

There is another set of Microsoft protocols that can complicate our mission. *MSRPC* is Microsoft's unique implementation of the *Distributed Computing Environment/Remote Procedure Calls (DCE/RPC)* protocol to support the use of distributed software that runs and communicates between different hosts.

Although SMB is used for file-oriented operations, MSRPC is typically used for operations such as resource management, user administration, logon, directory replication, and many others. While it can run over SMB, you will also find it running directly over TCP/IP. It is used quite extensively, is complex, and not well documented. We'll just skim the surface of MSRPC and will not examine its intricacies since it is so complex.

Since it uses a variety of formats, decoding these correctly is a challenge. MSRPC traffic may be found on many different ports. When sent over SMB, it can be found on TCP port 445. Even in a modern enterprise, you will find a significant amount of activity on the "legacy" port TCP/139. This form will require additional communication with the portmapper on port 135, which is used to find the dynamic port for an RPC service.

When run over port 445, SMB provides the authentication and doesn't require a separate session to find where a given RPC service is located. It uses a mechanism called a *named pipe* to communicate between the client and RPC server. Named pipes provide interprocess communications (IPC) between the client and server.

When operating over TCP/UDP directly (DCOM), RPC services register with the EPM (Endpoint Mapper) upon startup. The EPM must be queried at TCP/UDP port 135 to locate the RPC by a client. DCOM is a Distributed Common Object Model, providing an object-oriented RPC service interface used in many Microsoft applications, including MS Exchange. Under DCOM, SMB no longer acts as the intermediary. The RPC itself must handle any authentication and authorization requirements.

Of the three mechanisms on the slide, the DCOM model is the one that we're most interested in because it can really throw you for a loop. If you are ever trying to explain really odd-looking connections to what would normally be ephemeral ports on a Windows host, RPC is the most common explanation. Let's see how we can prove if that's what's happening. The HTTP/HTTPS option is provided to allow Microsoft administrators to tunnel Microsoft protocols through firewalls, which really isn't a good idea.

Example of MSRPC-over-TCP Session (DCOM)

- Notice the odd connection to the server on a high port.
 - If this is RPC, there should be an EPM (Endpoint Mapping Request).

No.	Source	Destination	Protocol	Src Port	Dst Port	Info
4...	192.168.61.20	192.168.50.25	TCP	62340	135	62340 → 135 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_P
4...	192.168.50.25	192.168.61.20	TCP	135	62340	135 → 62340 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1460 WS
4...	192.168.61.20	192.168.50.25	TCP	62340	135	62340 → 135 [ACK] Seq=1 Ack=1 Win=525568 Len=0
4...	192.168.61.20	192.168.50.25	DCER...	62340	135	135 Bind: call_id: 2, Fragment: Single, 3 context items: EPMv4 V3
4...	192.168.50.25	192.168.61.20	DCER...	135	62340	62340 Bind_ack: call_id: 2, Fragment: Single, max_xmit: 5840 max_re
4...	192.168.61.20	192.168.50.25	EPM	62340	135	135 Map request, DRSUAPI, 32bit NDR
4...	192.168.50.25	192.168.61.20	EPM	135	62340	62340 Map response, DRSUAPI, 32bit NDR
4...	192.168.61.20	192.168.50.25	TCP	62341	49155	62341 → 49155 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK
4...	192.168.50.25	192.168.61.20	TCP	49155	62341	62341 → 49155 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1460
4...	192.168.61.20	192.168.50.25	TCP	62341	49155	62341 → 49155 [ACK] Seq=1 Ack=1 Win=65536 Len=0
4...	192.168.61.20	192.168.50.25	TCP	62340	135	62340 → 135 [ACK] Seq=329 Ack=281 Win=525312 Len=0
4...	192.168.61.20	192.168.50.25	DCER...	62341	49155	49155 Bind: call_id: 2, Fragment: Single, 3 context items: DRSUAPI
4...	192.168.50.25	192.168.61.20	TCP	49155	62341	62341 → 49155 [ACK] Seq=1 Ack=1898 Win=65536 Len=0
4...	192.168.50.25	192.168.61.20	DCER...	49155	62341	62341 Bind_ack: call_id: 2, Fragment: Single, max_xmit: 5840 max_re

Let's have a look at how DCOM or MSRPC looks. The most common reason that it first comes to an analyst's attention is that you will detect some weird-looking connection to a high-numbered port on a Windows server. Your first suspicion is that the server perhaps is compromised or has some unauthorized code running on it. Before raising the incident alarm, however, we'd suggest that you should verify whether or not MSRPC is to blame. If it is, it should be very easy to detect. If an MSRPC is in use, then you should be able to find an **Endpoint Mapping Request** immediately before the connection to the high-numbered port.

✦ To view the output, enter the following on the command line:

wireshark smb.pcap

Hypothesis Confirmed

- TCP connection to port 135, the End Point Mapper (portmapper)
 - A request for "DRSUAPI" and a response that should say where the service is

No.	Source	Destination	Protocol	Src Port	Dst Port	Info
4...	192.168.61.20	192.168.50.25	TCP	62340	135	62340 → 135 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_P
4...	192.168.50.25	192.168.61.20	TCP	135	62340	135 → 62340 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1460 WS
4...	192.168.61.20	192.168.50.25	TCP	62340	135	62340 → 135 [ACK] Seq=1 Ack=1 Win=525568 Len=0
4...	192.168.50.25	192.168.50.25	DCER...	62340	135	Bind: call_id: 2, Fragment: Single, 3 context items: EPMv4 V3
4...	192.168.50.25	192.168.61.20	DCER...	135	62340	Bind_ack: call_id: 2, Fragment: Single, max_xmit: 5840 max_re
4...	192.168.61.20	192.168.50.25	EPM	62340	135	Map request, DRSUAPI, 32bit NDR
4...	192.168.50.25	192.168.61.20	EPM	135	62340	Map response, DRSUAPI, 32bit NDR
4...	192.168.50.25	192.168.61.20	TCP	49155	62341	49155 → 62341 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_P
4...	192.168.61.20	192.168.50.25	TCP	62341	49155	62341 → 49155 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1460
4...	192.168.61.20	192.168.50.25	TCP	62340	135	62340 → 135 [ACK] Seq=329 Ack=281 Win=525312 Len=0
4...	192.168.61.20	192.168.50.25	DCER...	62341	49155	Bind: call_id: 2, Fragment: Single, 3 context items: DRSUAPI
4...	192.168.50.25	192.168.61.20	TCP	49155	62341	49155 → 62341 [ACK] Seq=1 Ack=1898 Win=65536 Len=0
4...	192.168.50.25	192.168.61.20	DCER...	49155	62341	Bind_ack: call_id: 2, Fragment: Single, max_xmit: 5840 max_re

In fact, in this case, if we look immediately before the connection to port 49155, we see that there is a connection from the same host to port 135. Within that connection, we can see a "Map Request" occurring. Let's look at how the server responds to that request.

✦ To view the output, enter the following on the command line:

wireshark smb.pcap

Mystery Revealed

No.	Source	Destination	Protocol	Src Port	Dst Port	Info
4...	192.168.61.20	192.168.50.25	TCP	62340	135	62340 → 135 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_P
4...	192.168.50.25	192.168.61.20	TCP	135	62340	135 → 62340 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1460 WS
4...	192.168.61.20	192.168.50.25	TCP	62340	135	62340 → 135 [ACK] Seq=1 Ack=1 Win=525568 Len=0
4...	192.168.61.20	192.168.50.25	DCER...	62340	135	135 Bind: call_id: 2, Fragment: Single, 3 context items: EPMv4 V3
4...	192.168.50.25	192.168.61.20	DCER...	135	62340	62340 Bind_ack: call_id: 2, Fragment: Single, max_xmit: 5840 max_re
4...	192.168.61.20	192.168.50.25	EPM	62340	135	135 Map request, DRSUAPI, 32bit NDR
4...	192.168.50.25	192.168.61.20	EPM	135	62340	Map response, DRSUAPI, 32bit NDR
4...	192.168.61.20	192.168.50.25	TCP	62341	49155	62341 → 49155 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK
4...	192.168.50.25	192.168.61.20	TCP	49155	62341	62341 → 49155 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1460
4...	192.168.61.20	192.168.50.25	TCP	62341	49155	62341 → 49155 [ACK] Seq=1 Ack=1 Win=65536 Len=0
4...	192.168.61.20	192.168.50.25	TCP	62340	135	62340 → 135 [ACK] Seq=329 Ack=281 Win=525312 Len=0
4...	192.168.61.20	192.168.50.25	DCER...	62341	49155	49155 Bind: call_id: 2, Fragment: Single, 3 context items: DRSUAPI
4...	192.168.50.25	192.168.61.20	TCP	49155	62341	49155 → 62341 [ACK] Seq=1 Ack=1898 Win=65536 Len=0
4...	192.168.50.25	192.168.61.20	DCER...	49155	62341	62341 Bind ack: call_id: 2, Fragment: Single, max_xmit: 5840 max_re


```

Number of floors: 5
  ▶ Floor 1 UUID: DRSUAPI
  ▶ Floor 2 UUID: 32bit NDR
  ▶ Floor 3 RPC connection-oriented protocol
  ▶ Floor 4 TCP Port:49155
  ▶ Floor 5 IP:192.168.50.25
Return code: 0x00000000

```

In the response, we find our mystery port, 49155, along with the RPC name (DRSUAPI).

SANS



smb.pcap

SEC503: Intrusion Detection In-Depth 165

Selecting the **Map Response** packet and looking into the packet details, we find deep within the EPM response a **Tower Array** containing a **Tower Pointer** and, within that, **Floors**. Floor 1 will be the service that has been requested. Floor 4 will be the port number that the service is found on. Floor 5 will be the IP address where the service is running. Typically, this will be the same as the host sending the response, but in rare instances, an administrator may have configured the EPM to send hosts to another system for servicing.

Snort and SMB/RPC

Snort has an SMB/DCERPC preprocessor:

- Looks for simple anomalies
- Exposes some aspects of RPC communication
 - Create signatures for UUIDs, **flow** options for request combinations

SMB rules are still very rudimentary.

- Unless you need something from a narrow set of features, you need to specify byte strings or use **byte_test** (which we will cover in the next section).

```
alert tcp $HOME_NET any -> any any (msg:"EXPLOIT Possible ETERNALBLUE SMB
Exploit Attempt Stage 2/2 - Trans2 SUCCESS MultiplexID = 65 - MS17-010";
flow:to_server,established; content:"|FF|SMB|32 00 00 00 00|"; offset:4;
depth:9; content:"|41 00|"; distance:21; within:23; flowbits:isset,
SMB.v1.AndX.MID.64; classtype:trojan-activity; sid:5000075; rev:1;)
```

While Snort does have a preprocessor that supports SMB and DCERPC, the options that it makes available for detection are rather limited. The preprocessor primarily supports some anomaly detection and the ability to configure the type of operating systems that it should primarily be expected to see traffic from. It also exposes a few parameters that can be used with the **flow** keyword to look for the presence of various combinations of requests and the ability to identify an RPC by the UUID of that RPC.

Unfortunately, other obvious-seeming features are missing. For example, it would make sense for the preprocessor to expose the names of file shares and files being accessed, but it does not. The effect of this is that, even with this preprocessor, most SMB-related rules you will find in the rule feed will use a variety of byte strings to identify potentially malicious activities.

Section 3 Bootcamp Exercises!

Lab Workbook for Notes and Score Server for Questions
Workbook Exercises "SMB"

This page intentionally left blank.

COURSE RESOURCES AND CONTACT INFORMATION



AUTHOR CONTACT

David Hoelzer
dhoelzer@enclaveforensics.com
BEST CONTACT: The Slack Channel!



SANS INSTITUTE

11200 Rockville Pike, Suite 200
N. Bethesda, MD 20852
301.654.SANS(7267)



#SHOWMETHEPACKETS

<https://www.showmethepackets.com>



SANS EMAIL

GENERAL INQUIRIES: info@sans.org
REGISTRATION: registration@sans.org
TUITION: tuition@sans.org
PRESS/PR: press@sans.org

This page intentionally left blank.