

503.2

Fundamentals of Traffic Analysis Part II

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.



Fundamentals of Traffic Analysis: Part II

© 2021 David Hoelzer | All Rights Reserved | Version G01_01

This page intentionally left blank.

Table of Contents (1)	Page
Wireshark Display Filters	
➤ Display Filter Format	04
➤ Methods for Creating Display Filters	12
Wireshark Display Filters Exercises	30
Writing tcpdump Filters	
➤ Understanding tcpdump Filters	31
➤ Three Categories of Filters	46
➤ Miscellaneous Filter Topics	52
Writing tcpdump Filters Exercises	58
TCP	
➤ TCP Theory	59
➤ TCP Header Fields	63
SANS	SEC503: Intrusion Detection In-Depth 2

In this second section, we continue our tour up the TCP/IP model stack. We left you in suspense after Section 1, hanging from the intellectual cliff, no doubt with a sleepless night, wondering about what follows the IP layer. Now, we answer those questions and quell your anxiety by examining protocols associated with the transport layer. Specifically, the topics are TCP, UDP, and ICMP. It's much easier to examine these and the application layers that follow in Section 3 and beyond when you have more advanced knowledge of how the two primary tools, Wireshark and tcpdump, can selectively view or filter records.

Wireshark uses *display filters* to enable you to select what you want to view. Tcpdump uses *Berkeley Packet Filters (BPFs)*. Because Wireshark offers a more granular analysis of protocols, it has more selection criteria to specify packets for viewing. Tcpdump BPF syntax is more esoteric than and not nearly as flexible as Wireshark display filters. However, as we discussed in Section 1, tcpdump is fast and efficient at selecting packets, especially on large pcaps. It is essential for an analyst to have masterful command of both tools.

The demo files for Section 2 are found in the directory **/home/sans/demo-pcaps/Day2-demos**.

Table of Contents (2)	Page
➤ TCP Stimulus and Response	107
<i>TCP Exercises</i>	114
UDP	
➤ UDP Theory and Header Fields	115
➤ UDP Stimulus and Response	125
ICMP	
➤ ICMP Theory and Header Fields	128
<i>UDP-ICMP Exercises</i>	151
IPv6	
➤ IPv6 Fundamentals	153
Bootcamp	
➤ BPF and TCP Connections	175
<i>Section 2 Bootcamp Exercises</i>	184
SANS	SEC503: Intrusion Detection In-Depth 3

This page intentionally left blank.

Session 14.2 Packet/Network as a Second Language				
- 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 - Snow & 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

Fundamentals of Traffic Analysis: Part II

Display Filter Format

Methods for Creating Display Filters

- Wireshark Display Filters
- Writing tcpdump Filters
- TCP
- UDP
- ICMP
- IPv6

One of the most powerful features of Wireshark is the capability to selectively view packets that have a particular characteristic. This is especially useful when you have a lot of packets and want to find a specific packet or subset of packets that is of interest. Wireshark uses something known as a *display filter* to find one or more packets with a characteristic that you specify.

Wireshark display filters are totally different from and much easier to use than BPFs. They are used when you already have captured traffic and you want to display packets with a given trait. And they use a totally different syntax than BPF. It takes a while to get used to it, but Wireshark also facilitates the learning process because it has some default commonly used display filters and enables you to create your own by using its menus and lists or selecting a field/value to filter, all the while assisting you with an auto-complete feature, where applicable, to help you create a filter.

We will see that BPFs are primitive and more restricted in selecting packets and identifiable fields in packets, but they are superior when a large pcap needs to be examined for a field(s) of a predictable offset into a protocol header or data.

Capability to Select Traffic to Display

- Wireshark uses many different types of dissectors/protocol decoders.
- Allows individual field/values of given protocol to be examined and packet displayed if it matches selected filter
- More focused filtering capability than tcpdump
- Many ways to select/create display filters

Wireshark has a comprehensive set of protocol *dissectors* or *decoders*. This is one of the impressive features that distinguish Wireshark from many of the other open-source sniffers. A *protocol dissector/decoder* is software that analyzes the protocol, much like an application using that protocol would. Therefore, the protocol dissector or decoder must understand the protocol, follow it, and associate values found in the protocol with their respective fields.

For instance, if you have a protocol dissector for DNS, it can examine a DNS response for all the resource records returned. It can parse all the DNS resource records into components and show you the variable-length fields, such as a hostname. Protocol dissection is the most accurate means of finding content for an identifiable protocol field.

After a protocol is decoded, Wireshark can then expose both the fields and associated values to the user. This is a robust feature when you want to search for a specific value for a given field of the protocol.

Wireshark display filters represent high-level analysis, whereas tcpdump filters represent low-level analysis if you make the analogy of high-level and low-level programming languages. High-level represents more of an abstraction from the underlying language (say, assembly language) or packet data, whereas low-level is closer to the actual bits and bytes.

Understand that Wireshark display filters are different from the BPF capture filters that select packets to be captured when Wireshark is capturing network traffic. We will cover BPF filters in the next section.

Display Filter Format

- Simple indicator of presence of protocol/field:
 `dns`
 `ftp.response`
- Indicator of condition:
 `ip.fragment.overlap`
 `udp.checksum.bad`
- Field name – comparison – value:
 `ip.src == 192.168.11.65`
 `ipv6.fragment.offset > 0`

Display filters come in several formats, depending upon what you are seeking. Display filters can indicate the presence of a given protocol or field/value. For instance, the filter **dns** finds what Wireshark believes to be DNS records. The **ftp.response** filter looks for a source port associated with FTP.

Wireshark can also do some analysis to make determinations. For instance, the **ip.fragment.overlap** condition results when two fragments in the same fragment train have overlapping offsets. Without Wireshark's discovery and detection of this condition during dissection, it would not be possible for a user to create a filter that performed this comparison. Most display filters at your disposal are incapable of the complex logic involved in comparing multiple records. They can compare fields and values, but not one record with another. But a limited number of canned display filters can perform such comparisons.

udp.checksum.bad exposes all UDP packets that have invalid checksums. This is something that is computed or discovered when the UDP packet is dissected, before applying any filters. Again, this is something too complex for a user-created Wireshark display filter because there is no feature to apply arithmetic formulas on selected fields.

Probably the type of filter you will use most has the format of specifying a field, an operator, and a value. The filter **ip.src == 192.168.11.65** looks for any record with a source IP of 192.168.11.65. The filter **ipv6.fragment.offset > 0** searches for any IPv6 fragment with an offset of greater than 0.

Comparison Operators

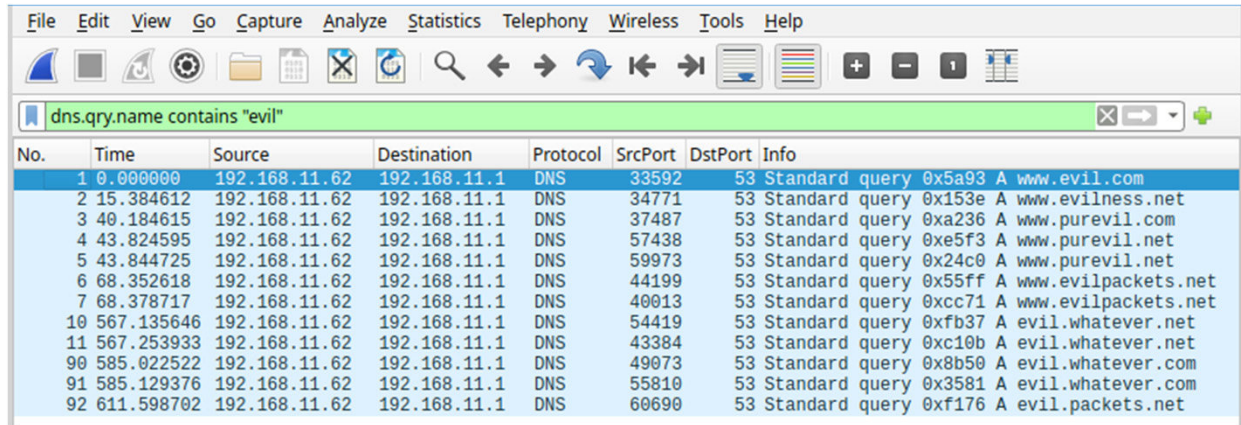
Comparison	Operator
Equal	==
Not equal	!=
Greater than	>
Less than	<
Greater than or equal	>=
Less than or equal	<=
Contains	contains
Regular expression	matches

Comparison	Operator
And	&& / and
Or	/ or
Not	! / not

Wireshark offers a typical set of comparison operators, as shown here. This is a sample of the most commonly used comparison operators; you can find more in the documentation on the Wireshark website at <http://www.wireshark.org/docs>. Look for the section on display filters in the *Wireshark User's Guide*.

Most of these operators are straightforward and similar to expressions you may use in other languages. Some need more explanation in the following slides.

Contains Operator



The screenshot shows the Wireshark network protocol analyzer. The filter bar at the top contains the expression `dns.qry.name contains "evil"`. Below the filter bar, a list of 12 DNS query packets is displayed, all of which contain the string "evil" in their query names.

No.	Time	Source	Destination	Protocol	SrcPort	DstPort	Info
1	0.000000	192.168.11.62	192.168.11.1	DNS	33592	53	Standard query 0x5a93 A www.evil.com
2	15.384612	192.168.11.62	192.168.11.1	DNS	34771	53	Standard query 0x153e A www.evillness.net
3	40.184615	192.168.11.62	192.168.11.1	DNS	37487	53	Standard query 0xa236 A www.purevil.com
4	43.824595	192.168.11.62	192.168.11.1	DNS	57438	53	Standard query 0xe5f3 A www.purevil.net
5	43.844725	192.168.11.62	192.168.11.1	DNS	59973	53	Standard query 0x24c0 A www.purevil.net
6	68.352618	192.168.11.62	192.168.11.1	DNS	44199	53	Standard query 0x55ff A www.evilpackets.net
7	68.378717	192.168.11.62	192.168.11.1	DNS	40013	53	Standard query 0xcc71 A www.evilpackets.net
10	567.135646	192.168.11.62	192.168.11.1	DNS	54419	53	Standard query 0xfb37 A evil.whatever.net
11	567.253933	192.168.11.62	192.168.11.1	DNS	43384	53	Standard query 0xc10b A evil.whatever.net
90	585.022522	192.168.11.62	192.168.11.1	DNS	49073	53	Standard query 0x8b50 A evil.whatever.com
91	585.129376	192.168.11.62	192.168.11.1	DNS	55810	53	Standard query 0x3581 A evil.whatever.com
92	611.598702	192.168.11.62	192.168.11.1	DNS	60690	53	Standard query 0xf176 A evil.packets.net



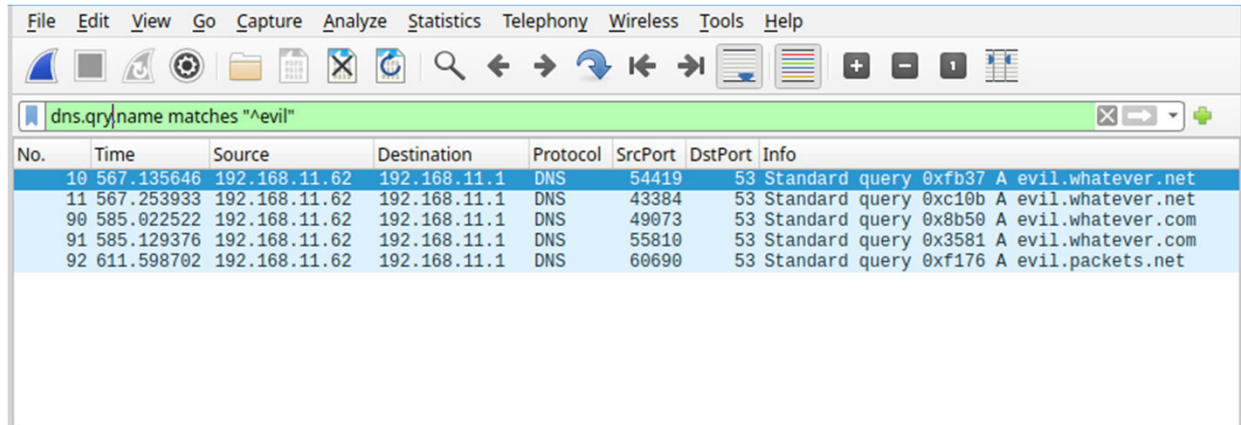
The **contains** comparison operator can be used for a search where you want to find a particular value in a specific field. For instance, the filter here, `dns.qry.name contains evil`, focuses on the query name in a DNS record looking for an occurrence of the string value "evil" and displays all the records found. There are plenty of "evil" packets!



To follow along with the slide, enter the following on the command line:

wireshark evildns-queries.pcap

Matches Operator



No.	Time	Source	Destination	Protocol	SrcPort	DstPort	Info
10	567.135646	192.168.11.62	192.168.11.1	DNS	54419	53	Standard query 0xfb37 A evil.whatever.net
11	567.253933	192.168.11.62	192.168.11.1	DNS	43384	53	Standard query 0xc10b A evil.whatever.net
90	585.022522	192.168.11.62	192.168.11.1	DNS	49073	53	Standard query 0x8b50 A evil.whatever.com
91	585.129376	192.168.11.62	192.168.11.1	DNS	55810	53	Standard query 0x3581 A evil.whatever.com
92	611.598702	192.168.11.62	192.168.11.1	DNS	60690	53	Standard query 0xf176 A evil.packets.net



You might be wondering what the difference is between the operators **contains** and **matches**. The **matches** operator allows *regular expression* syntax to be used to specify a more complex description of content to find.

The **matches** filter seen on the slide selects only records with "evil" at the beginning of the DNS query name. The **matches** operator is not well documented. My advice is to experiment with regular expression syntax for matches that you'd like to find. Additionally, while we can use regular expressions in Wireshark, be cautious of becoming overly reliant on them. Regular expressions are wonderful when you are working with an offline packet capture, but if you are attempting to create real-time signature for a monitoring system, regular expressions are *very* expensive computationally. In this class, we urge you to never think of things in terms of a handful of packets; always think of things in terms of millions or billions of packets to ensure that we are always using optimal approaches that translate to our enterprise.



The "^" (carat) notation is used in regular expressions to indicate that the text that follows is found at the beginning of a line.

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

wireshark evildns-queries.pcap

What's the Difference Between "Contains" and "Find Packet"

Contains:

- Limits your view to *only* packets that match your expression
- Especially useful to see large numbers of matches while isolating them from others

Find Packet:

- Does not change your current view of packets
- Jumps to the next packet that matches your search criteria
- Especially useful for seeing data in context

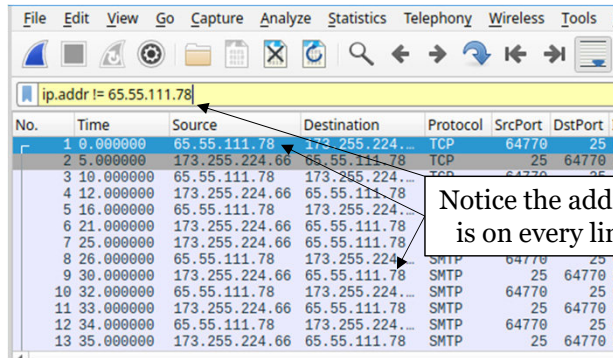
In the first coursebook, you learned about the Wireshark feature **Find Packet**. You might wonder why such a feature exists if we can apply a filter using the keyword **contains**. In fact, both are useful in their own way.

The **contains** keyword is especially useful when you have a large number of packets, and you wish to see all of the packets that match the searched-for string *at the same time*. This can be very useful when looking for patterns of activity. In this case, the context of each match (surrounding packets) isn't what you are currently focused on.

On the other hand, if you are more interested in the surrounding context (surrounding packets) related to the item you are searching for, the **Find Packet** feature makes more sense to use. While this will require that you search repeatedly, perhaps to find the packets of particular interest, it allows you to still see all of the surrounding packets in the **packet list** panel at the top.

A Common Confusion

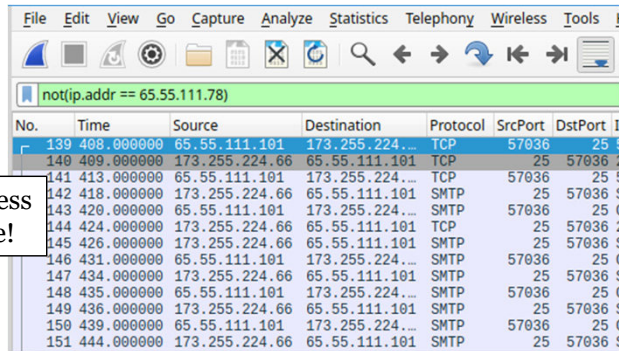
Likely Incorrect



No.	Time	Source	Destination	Protocol	SrcPort	DstPort
1	0.000000	65.55.111.78	173.255.224.66	TCP	64770	25
2	5.000000	173.255.224.66	65.55.111.78	TCP	25	64770
3	10.000000	65.55.111.78	173.255.224.66	TCP	64770	25
4	12.000000	173.255.224.66	65.55.111.78	TCP	25	64770
5	16.000000	65.55.111.78	173.255.224.66	TCP	64770	25
6	21.000000	173.255.224.66	65.55.111.78	TCP	25	64770
7	25.000000	173.255.224.66	65.55.111.78	TCP	25	64770
8	26.000000	65.55.111.78	173.255.224.66	TCP	64770	25
9	30.000000	173.255.224.66	65.55.111.78	TCP	25	64770
10	32.000000	65.55.111.78	173.255.224.66	TCP	64770	25
11	33.000000	173.255.224.66	65.55.111.78	TCP	25	64770
12	34.000000	65.55.111.78	173.255.224.66	TCP	64770	25
13	35.000000	173.255.224.66	65.55.111.78	TCP	25	64770

Notice the address
is on every line!

Correct



No.	Time	Source	Destination	Protocol	SrcPort	DstPort
139	408.000000	65.55.111.101	173.255.224.66	TCP	57036	25
140	409.000000	173.255.224.66	65.55.111.101	TCP	25	57036
141	413.000000	65.55.111.101	173.255.224.66	TCP	57036	25
142	418.000000	173.255.224.66	65.55.111.101	TCP	25	57036
143	420.000000	65.55.111.101	173.255.224.66	TCP	57036	25
144	424.000000	173.255.224.66	65.55.111.101	TCP	25	57036
145	426.000000	173.255.224.66	65.55.111.101	TCP	25	57036
146	431.000000	65.55.111.101	173.255.224.66	TCP	57036	25
147	434.000000	173.255.224.66	65.55.111.101	TCP	25	57036
148	435.000000	65.55.111.101	173.255.224.66	TCP	57036	25
149	436.000000	173.255.224.66	65.55.111.101	TCP	25	57036
150	439.000000	65.55.111.101	173.255.224.66	TCP	57036	25
151	444.000000	173.255.224.66	65.55.111.101	TCP	25	57036



The **not** or **!** operator negates a particular condition. It seems straightforward; however, one of the caveats offered in the documentation is the use of the **!=** operator.

The example of **ip.addr != 65.55.111.78** is an incorrect filter to find all packets where the IP address is not 65.55.111.78. The interpretation of this by Wireshark is that the packet must contain an IP address that is different from 65.55.111.78. Each packet has both a source and destination IP address, and either the source or destination IP address will be different, exposing all records in the capture.

The proper way to compose the filter is **not(ip.addr == 65.55.111.78)**, which is translated as neither the source nor destination IP address has a value of 65.55.111.78 (note that you could have replaced the word **not** in this expression with **!**). Just be aware of this issue when you want to negate some value or expression.

Notice that Wireshark is trying to help you. When the filter bar is red, it means that the filter is invalid. When the bar is green, it means that the filter is valid. When the bar is yellow, it means, "I can do that, but that's probably not what you want."

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

wireshark phishing-attack.pcap

Section 14.2 Packet/Networks as a Second Language	Section 14.3 Scapy	Section 14.4 Zeek	Section 14.5 Wireshark	Section 14.6 tcpdump
• 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
Prerequisite: Networking 101, Packet World for configuration & capture				

Fundamentals of Traffic Analysis: Part II

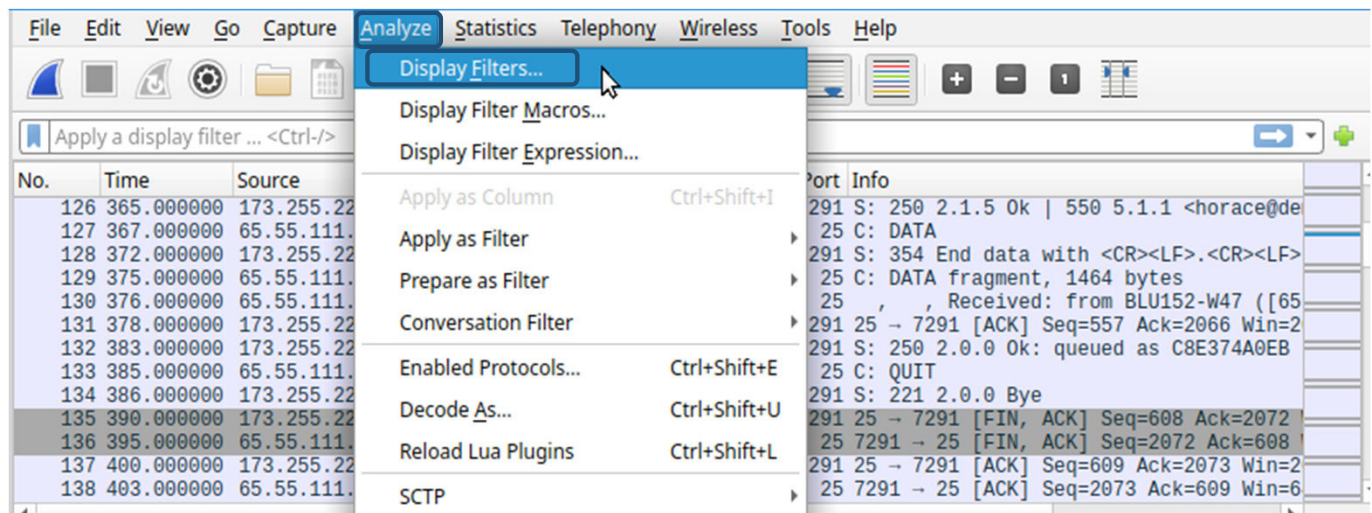
Display Filter Format

Methods for Creating Display Filters

- Wireshark Display Filters
- Writing tcpdump Filters
- TCP
- UDP
- ICMP
- IPv6

This page intentionally left blank.

Selecting One of Wireshark's Canned Filters



SANS



canned-filters.pcap

SEC503: Intrusion Detection In-Depth 13

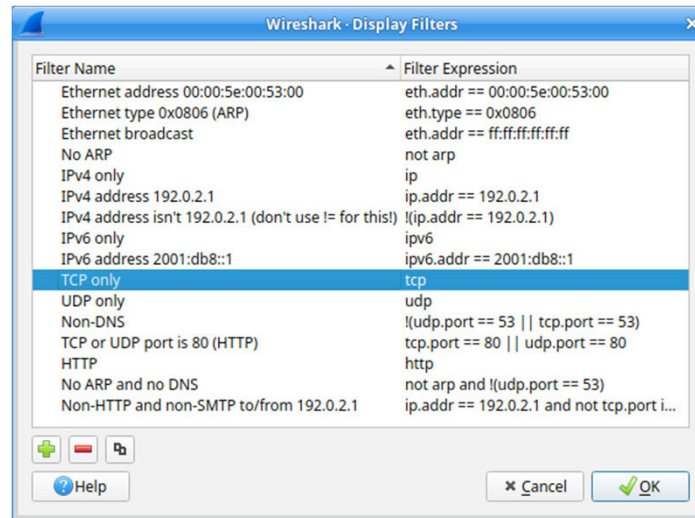
Some of the most common and simple display filters are available as default or canned filters. You can access these by navigating to the **Analyze → Display Filters** menu/selection.



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

wireshark canned-filters.pcap

Sample Canned Display Filters



As you can see, some basic supplied filters are available. The menu shows you the descriptive name of the filter in the left column and the details of the filter itself in the right column. Buttons are available to allow you to add or remove filters, in addition to copying a filter so that you could potentially modify the copy. With your duplicate filter created, you can now double-click on it to edit the name or the filter.

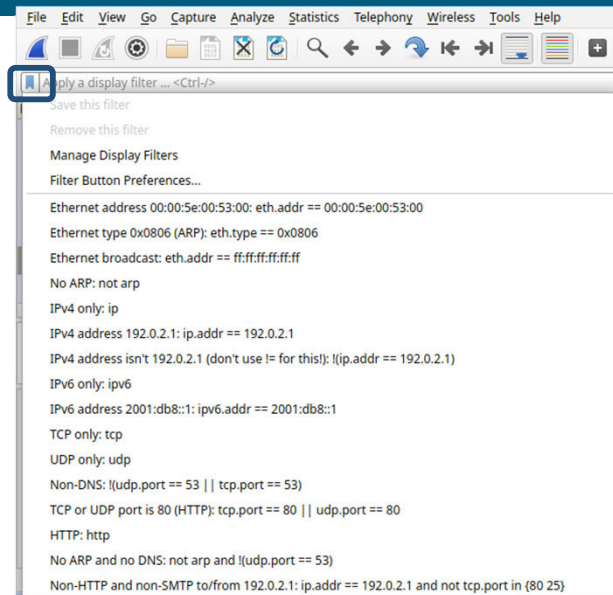
You might intuitively assume that you can use this dialog box to select a filter that you would like to apply. While this was true in past versions of Wireshark, this capability has mysteriously disappeared. It's difficult to tell if this is a bug or if it was intentional, but we suspect this feature will return in the future.



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

wireshark canned-filters.pcap

Alternative Canned Filters



SANS



canned-filters.pcap

SEC503: Intrusion Detection In-Depth 15

Alternatively, you can also pull up the preinstalled filters by clicking on the blue ribbon, which you can think of as a "Favorite Filters" or "Filter Bookmarks" icon. Clicking on that ribbon button allows us to see and select any of the preexisting filters that were available through the display filters option in the Analyze menu.

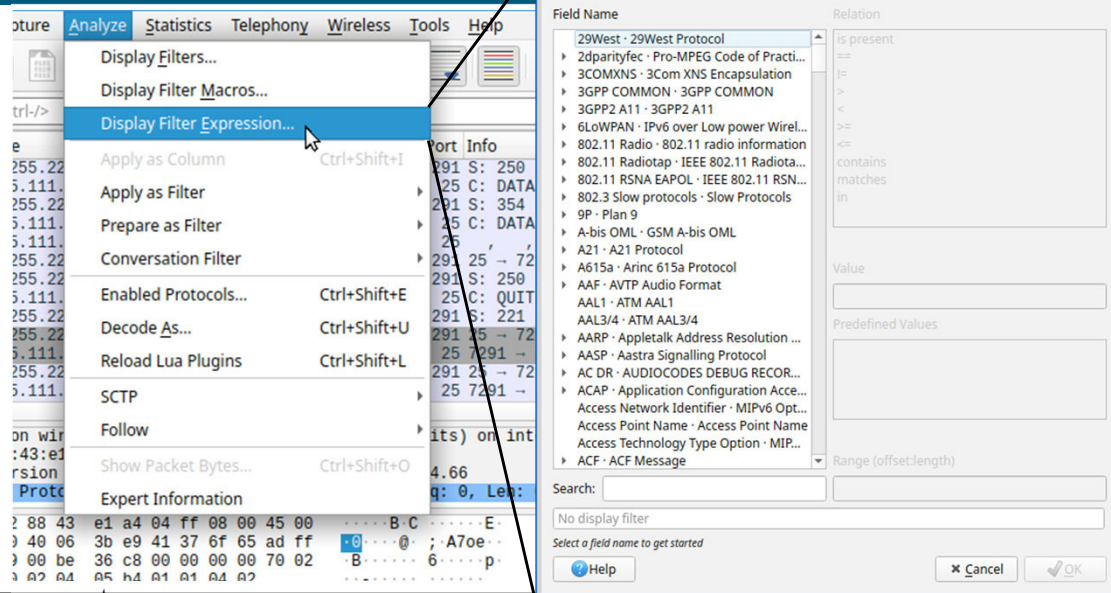


Notice that there is also a "Manage Display Filters" option. This will open the same dialog box for display filter management that we saw on the previous page. As a reminder, the menu option Analyze→Manage Display Filters can be used to access this dialog as well.

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

wireshark canned-filters.pcap

Expression Menu



SANS



phishing.pcap

SEC503: Intrusion Detection In-Depth 16

What if you want to create a filter but you don't really know where to start? For example, you know that you want to filter on a particular protocol, but you aren't sure how Wireshark notates that protocol. In the "Analyze" menu, you might want to try to use the "Display Filter Expression" option. When you select this option, a dialog box will open, as pictured in the slide. This dialog box has a list of all the different protocols that Wireshark understands. You can either scroll through this list to peruse the protocols or look for the one you are interested in, or if you have some idea what Wireshark might call it, you can use the "Search" box toward the bottom left. Be warned, however, that searching for something like "TCP" will still return dozens of protocols, so expect to do some scrolling and reading to find exactly what you want. Each of these protocols can be expanded to reveal different fields for comparison or some conditions associated with the selected protocol.



Filter Expression Menu

The dialog box shows the 'Field Name' list on the left, with 'tcp.hdr_len · Header Length' selected. The 'Relation' list on the right shows '==' selected. The 'Value' field is set to '20'. The 'Search' field contains 'tcp'. The 'Predefined Values' and 'Range (offset:length)' sections are empty. The 'OK' button is highlighted.

Field Name	Relation	Value
tcp.flags · Flags	tcp.reset	
tcp.flags.ack · Acknowledgment	==	
tcp.flags.cwr · Congestion Window R...	>	
tcp.flags.ecn · ECN-Echo	<	
tcp.flags.fin · Fin	>=	
tcp.flags.ns · Nonce	<=	
tcp.flags.push · Push	in	
tcp.flags.res · Reserved		
tcp.flags.reset · Reset		
tcp.flags.str · TCP Flags		
tcp.flags.syn · Syn		
tcp.flags.urg · Urgent		
tcp.hdr_len · Header Length		20
tcp.len · TCP Segment Len		
tcp.nop · 4 NOP in a row - a router ...		
tcp.nxtseq · Next sequence number		
tcp.option_kind · Kind		
tcp.option_len · Length		
tcp.option.len.invalid · Invalid lengt...		
tcp.options · TCP Options		
tcp.options.cc_value · TCP CC Option		

Search: tcp

tcp.hdr_len == 20

Click OK to insert this filter

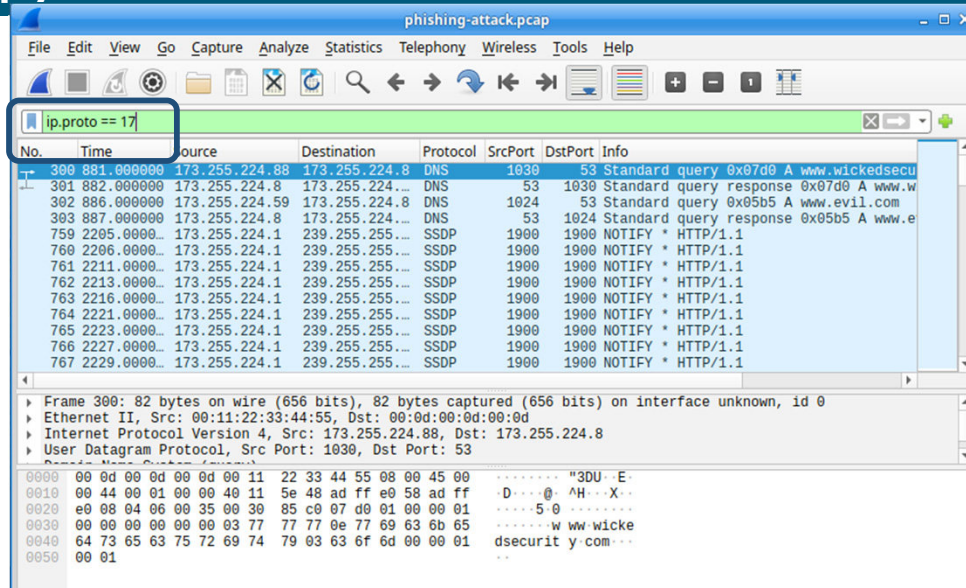
Help Cancel OK



Let's imagine that we want to find TCP packets with a header length of 20. First, we find the TCP protocol. Once we have found that, we can click to open that protocol and locate the field we are interested in. After you find the field you want, the "Relation" pane in the dialog box becomes active, allowing you to select a relational operator. Once you select a relation, the "Value" field becomes active, allowing you to define the value you are looking for. Note that Wireshark is cognizant of the appropriate relations that apply for the chosen field. What this means is that since we are examining a field that can only contain a number, Wireshark will not present you with the "contains" or "matches" operators.



Supply Your Own Filter



SANS



phishing-attack.pcap

SEC503: Intrusion Detection In-Depth 18

When you are comfortable with the syntax and the choices in Wireshark, you may find that it is more efficient to enter some of your own display filters. If you know you want to see UDP records only, you can create a filter to look for IP protocol 17 by entering the filter `ip.proto==17`. When this filter is applied, only UDP records display. Of course, there is more than one way to do almost anything in Wireshark. For example, this filter could be simplified to just `udp`, which checks for the presence of a UDP packet. Why would you use this over examining the IP protocol field?



What if our packets contained IPv4 and IPv6 packets? If we use the `ip.proto==17` filter, we will *only* see IPv4 packets that contain UDP. This might be our goal! However, if we want to see *all* UDP packets regardless of how they were transported, the `udp` filter would show us all UDP datagrams, even if they are being carried inside of some other protocol (like IPv6 or ICMP... We will talk about why they might be inside of ICMP packets later in this book).

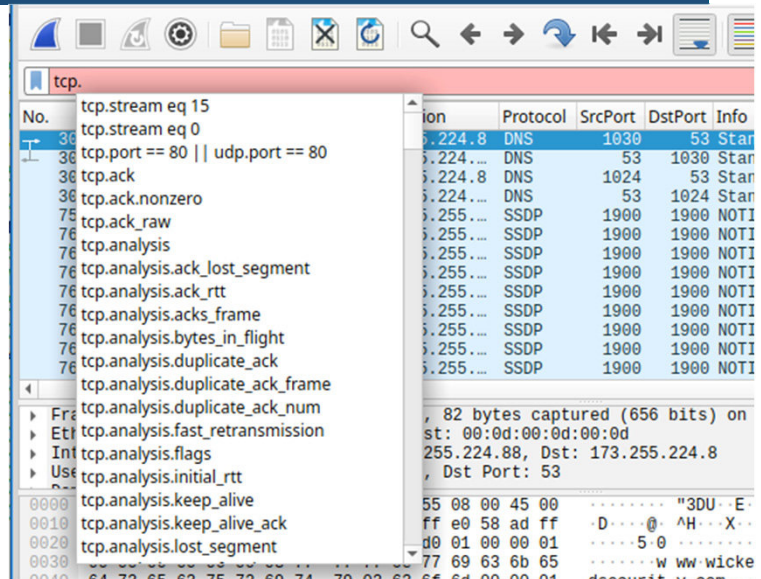
Auto-Complete Function

Autocomplete gives context:

- Start typing the protocol.
- You will see completion options.
- If you hit the period, it refines the options available.

It is worth your time to explore the options:

- Especially the "analysis" options!



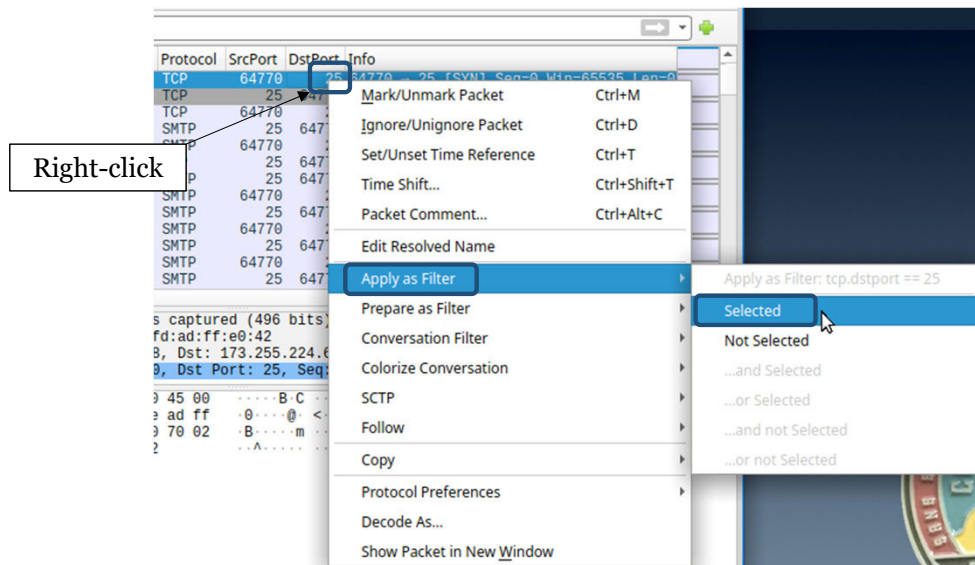
Wireshark has a very helpful feature known as *auto-complete*. Whenever you manually enter a filter that designates a particular protocol that you are able to remember but cannot remember a given field name for that protocol, simply type a period after the protocol. This is the catalyst for Wireshark to bring up all the like-named fields of the one you entered. Whenever you enter a unique portion of a field name, Wireshark completes it for you. For instance, enter **tcp.h** in the filter input, and Wireshark completes it as **tcp.hdr_len** because that is the only field name or condition that it knows of with **tcp.h**.

Even if you already know these protocols well, we would strongly recommend that you take some time to peruse the various completions that are available. Some of the most interesting completions have the word "analysis" in them. What are those?

If you are looking at something like "tcp.ack", you are performing a comparison against a specific field in the header. Compare that to "tcp.analysis.ack_rtt". If you get out your TCP/IP quick reference and look at the TCP header, you will find that there is no field named "ack_rtt". What is this, then? This is the TCP protocol dissector within Wireshark performing some expert analysis for you! For example, this specific analysis option allows you to look for ACK packets where the round-trip time from when the TCP segment was sent to the arrival of the corresponding ACK is less than, equal to, or greater than some value that you choose. There is nothing in the packet that tells you this number; this is a high-level analysis that Wireshark is performing.

There are many other analysis options that can be very useful for network diagnostics, troubleshooting, threat hunting, and otherwise understanding network behaviors.

Apply as Filter by Packet Pane Record Column



Perhaps the most powerful convenience feature related to filtering that Wireshark supports is the ability to click on a field and instantly create a filter for that field. As a general rule, if you can right-click on something in Wireshark, you can create a filter from that point. For example, if we are looking at the packet list, we can right-click on any of the fields that are visible and choose "Apply as Filter." From here, we can choose to generate a filter that shows only packets that match the value we've clicked on.

Notice that there are also options to allow us to exclude packets with the selected value from our view. We can also modify the filter that is currently in our filter expression field by anding or oring the selected value with the existing filter.

In the slide, we have right-clicked on the TCP destination port field where the value is 25.



Apply as Filter by Field and Value

The image shows a Wireshark packet capture of a phishing attack. The top pane displays a list of packets. Packet 300 is selected, showing a DNS query from 173.255.224.88 to 173.255.224.88. The middle pane shows the details of packet 300, including Ethernet II, Internet Protocol Version 4, User Datagram Protocol, and Domain Name System (query). The bottom pane shows the raw data of the packet. A right-click context menu is open over the 'Transaction ID: 0x07d0' field in the DNS header. The menu options are: Expand Subtrees (Shift+Right), Collapse Subtrees (Shift+Left), Expand All (Ctrl+Right), Collapse All (Ctrl+Left), Apply as Column (Ctrl+Shift+I), Apply as Filter (Selected), Prepare a Filter (Not Selected), and Conversation Filter (...and Selected).

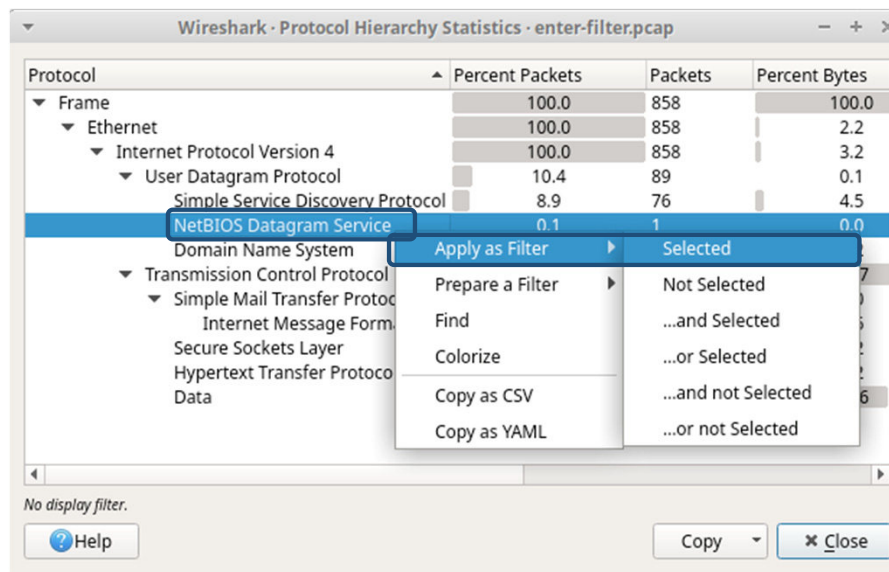
No.	Source	Destination	Protocol	Src Port	Dst Port	Info
299	65.55.111.101	173.255.224.66	TCP	28444	25	28444 → 25 [ACK] Seq=2007 Ack=281 Win=65256 Len=
300	173.255.224.88	173.255.224.88	DNS	1030	53	Standard query 0x07d0 A www.wickedsecurity.com
301	173.255.224.8	173.255.224.88	DNS	53	1030	Standard query response 0x07d0 A www.wickedsecu
302	173.255.224.59	173.255.224.8	DNS	1024	53	Standard query 0x05b5 A www.evil.com
303	173.255.224.8	173.255.224.59	DNS	53	1024	Standard query response 0x05b5 A www.evil.com A
304	173.255.224.59	10.100.100.111	TCP	48516	9999	48516 → 9999 [SYN] Seq=0 Win=5840 Len=0 MSS=1460
305	10.100.100.111	173.255.224.59	TCP	9999	48516	9999 → 48516 [SYN, ACK] Seq=0 Ack=1 Win=5792 Len=
306	173.255.224.59	10.100.100.111	TCP	48516	9999	48516 → 9999 [ACK] Seq=1 Ack=1 Win=5856 Len=0 TS



Remember that we said that if you can right-click on something, you can almost always choose to apply this as a filter. For example, here we have located a DNS query (this query is packet 300 in our phishing-attack.pcap file). In the packet details pane, we have opened the DNS header, allowing us to see the "Transaction ID" and other fields. You can right-click on any header value you're interested in from within packet details to build or add to a filter. In our case, we have right-clicked on the transaction ID value 0x07d0 and chosen "Apply as Filter" and "Selected."



Apply as Filter on Statistics Output



This ability to right-click and create a filter is not limited to the main Wireshark window. For example, you might open the Protocol Hierarchy view (from the Statistics menu) to see which protocols are present. Perhaps one of the protocols stands out as interesting or as unexpected. Rather than having to go back to the main Wireshark window to find the packets where that protocol is in use, you can simply right-click on the protocol value in the Protocol Hierarchy window and create a filter! This can really speed up your analysis of a packet capture when you're trying to zero in on interesting activities.



Prepare as Filter

When handling larger captures, **Apply as Filter** isn't always a great choice.

- Requires Wireshark to reprocess the filter every time
- Can be very time consuming

Prepare a Filter doesn't *apply* the filter.

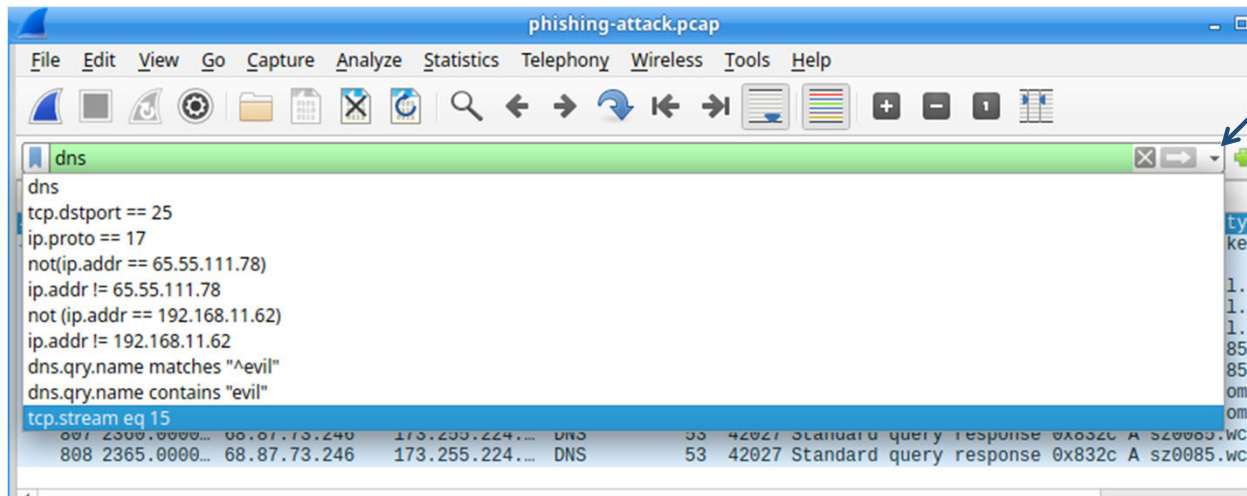
- Adds to or replaces the text in the filter box
- You can continue to add more and more criteria.
- When you have the entire filter, you can apply the filter one time.

You may have noticed that there is another option available below **Apply as Filter**. The **Prepare as Filter** and related sub-options are quite useful, especially when dealing with a larger capture file.

Whenever you use the **Apply as Filter** option, it will immediately apply the filter to the packet capture, as the name implies. The problem is that this can be very time consuming with a large packet capture. Wireshark must reprocess the file every time you apply a new filter!

On the other hand, the **Prepare a Filter** option still creates the same filter that the **Apply** option would, but it only puts it into the filter bar, either adding to what is there already or replacing it completely. This allows you to continue to add more and more filter options before finally applying the filter. The benefit is that you only perform the costly reprocessing option one time!

Retrieve Previous Filters



Sometimes we see someone who has taken a lot of time to carefully craft a useful filter and is reluctant to erase it to check for some other activity. They may fear that they will be unable to re-create the filter or that re-creating the filter will be very time consuming. Fortunately, Wireshark has a feature that can help.

A time-saving feature is the capability to retrieve some of the recent filters that you've used. This filter history is maintained by Wireshark automatically. To see the history, click on the down arrow next to the filter expression input field. Once we have pulled this down, we can click the filter that you want to use.

These display filters are stored in the `.config/wireshark/recent_common` file in the user's home directory. Names of recently used pcap files are stored in that file, too. They can be used when you select the **File→Open Recent** option or you invoke Wireshark from the command line without supplying a pcap. Both display filter and pcap lists have a limited number of entries, and the last one is deleted to make room for a new one when the limit is reached.

This means that we can even carry a display filter across separate runs of Wireshark. We can even share these filters easily with our colleagues!

Extracting Evidence

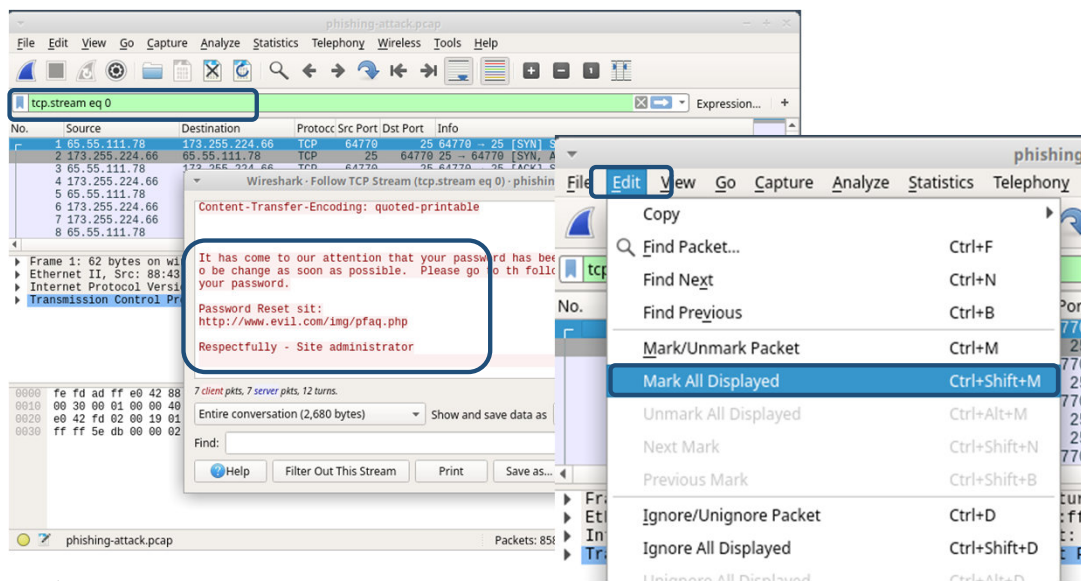
- The best evidence is the original packet capture.
 - A massive packet capture might not be the best thing to include with your report.
- Would this be useful to your process?
 - Within the larger packet capture for a phishing attack, for instance:
 - Identify the packets showing the phishing email.
 - Identify the packets showing that the user retrieved the email.
 - Identify the packets showing the DNS resolution when the user clicks.
 - Identify the packets showing the malware download after the click.
 - Identify the packets showing the malware exfiltrating data.
 - Extract and save all of *those* packets into a file.

Before we introduce the next feature in Wireshark that we'd like to bring to your attention, let's first lay out why this feature is useful. If you are engaged in an incident investigation, the relevant packet captures should be an important piece of evidence that you will analyze. While the complete and unfiltered packet capture is the best evidence you have, a massive packet capture can be unwieldy to pass between investigators and to include with your report. Can we do better?

What if we were able to analyze the larger packet capture and, while doing so, to somehow mark packets related to our investigation? We're not speaking directly about filtering here. Writing a filter that would take in all of the aspects of a phishing attack, for instance, would be large and complex. Instead, we'd like to somehow identify all of the packets that are related to the incident being investigated and then extract only those packets into another file.

The original packet capture remains the actual evidence, but the extracted file would be very useful for review with the report and to use to show lawyers and other non-packet nerds. Can we do such a thing with Wireshark? Absolutely!

Extract Relevant Packets – Phishing Email

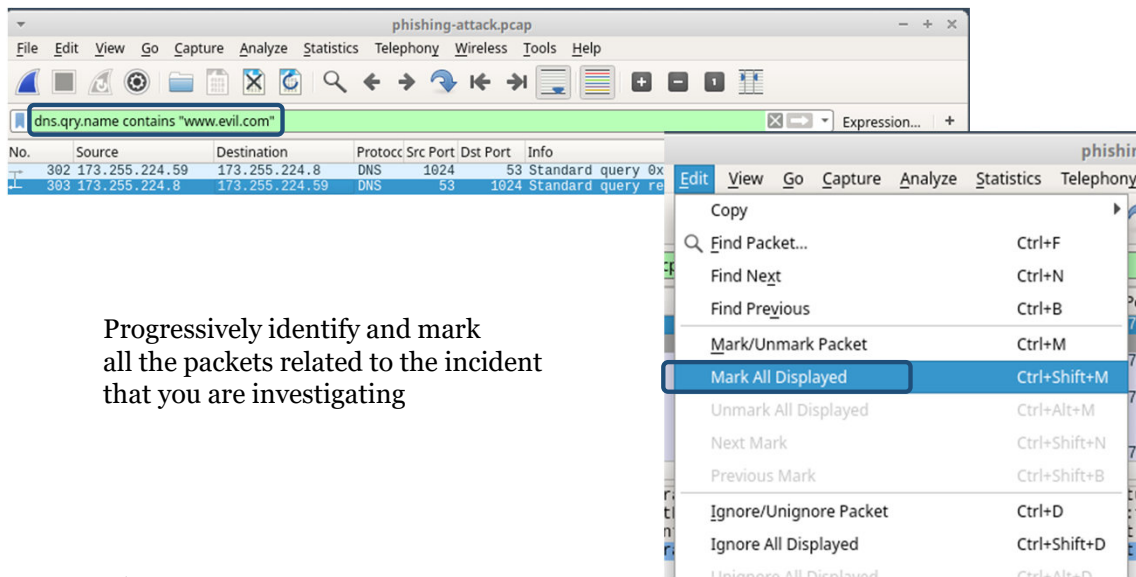


Let's walk through how you would mark and then export packets. The first task is to find packets that are interesting for you. In the example we are using, that of a phishing attack, we will start with a discovered phishing email. As you can see in the slide, we have identified a stream that contains the phishing email and can see that the email contains a URL pointing to a site where, likely, malicious code is being hosted.

When we followed this stream, Wireshark automatically created a filter, seen in the filter bar, limiting our display to this TCP stream. We can use this to our advantage. Since the first set of packets that we are interested in are the only things being displayed, we can use the **Edit -> Mark All Displayed** option. This option will not modify the packet capture! This simply marks the packets in memory so that Wireshark remembers that we are interested in them.

Notice that this same menu includes an option to allow us to mark a single packet. This can be useful when we have not bothered to create a display filter and wish to simply mark one or two packets. If you wish to mark a large number of packets, it is usually best to create a display filter first and then use the Mark All Displayed option.

Extract Relevant Packets – The DNS Query

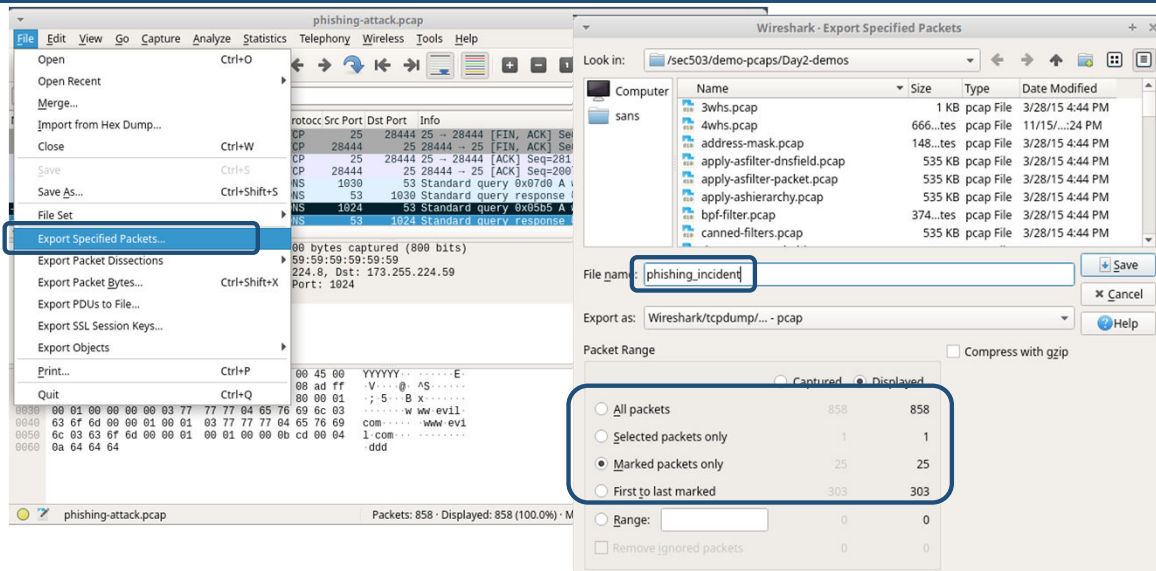


Progressively identify and mark all the packets related to the incident that you are investigating



If we simply wanted to export the phishing email, we actually could have done that using the **File** menu. In our case, however, we want to collect *all* the packets related to this incident. In the slide here, you can see that we have created a display filter that seeks to find the DNS query and response related to the domain name that was in the URL from the phishing email. Once we have found this query and the response, we again choose to use the **Edit** -> **Mark All Displayed** option to add these packets to our marked packets.

Extract Relevant Packets – Export Marked Packets



Imagine that we continue identifying the various packets related to the activities involved in this incident. When we finish, we can now export all of these marked packets as a single capture. To do so, use the **File -> Export Specified Packets...** option. This will open the dialog box that you see on the right-hand side of the slide. In this dialog box, you can choose, using the bottom section, precisely what it is you would like to save. In our case, we wish to export **Marked packets only**. In this case, since we do not currently have a display filter active, we could use either the **Captured** or the **Displayed** option. In either case, you can see that the columns to the right indicate how many packets would be exported, which can be useful in helping us to select the correct option.

With these options selected, the only remaining task is to select a filename for our packets! Now we have all of the packets related to the incident being analyzed in a single, self-contained location.

Wrap-Up of Wireshark Display Filters

- Frequently used feature to assist in finding packets with a given characteristic
- Allows for easy and more granular inspection than BPF
- Many different ways to generate a display filter
- Don't confuse with Wireshark capture filters

In conclusion, the Wireshark display filters feature is one of the most versatile and often-used features of Wireshark. These filters are extremely adept at finding packets with an attribute you'd like to examine in one or more packets. As you'll see, they are much simpler to create than BPF expressions and enable you to specify a field of interest easily. And Wireshark has more comparison operators than BPF.

Wireshark offers many intuitive methods for generating a display filter. You can create your own in the filter entry, use Wireshark's expression menu to select a protocol/field along with a comparison operator and value, recall a previous filter, and use many fields on different output displays, packets, or statistics to invoke the **Apply as Filter** feature. Wireshark has anticipated and accommodated a variety of search methods for ease of use.

Wireshark Display Filters Exercises

Workbook exercises "Wireshark Display Filters"

This page intentionally left blank.

Section 14.2 Packet Sniffing as a Second Language

• 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

Fundamentals of Traffic Analysis: Part II

Understanding tcpdump Filters

Three Categories of Filters

Miscellaneous Filter Topics

- Wireshark Display Filters
- Writing tcpdump Filters
- TCP
- UDP
- ICMP
- IPv6

Tcpdump filters, also known as *Berkeley Packet Filters (BPFs)*, are necessary to selectively gather/read/write records of network traffic. Many times, you want to process packets with a desired characteristic. This can be done by telling tcpdump which packet fields and values to process by using filters. This is most helpful if you have a large collection of libpcap records that you want to examine for some characteristic.

For instance, say that you see a SYN scan of a TCP port coming into your network, and you've captured the scan activity and anything else that traversed the network at the same time. An important analysis would be to determine if any of the scanned hosts responded that they listened on the scanned ports. It would be possible to do this using tcpdump by filtering packets returned from a scanned port that have the SYN and ACK flags set. These flags indicate that a server is listening on those ports.

Most of the time, writing tcpdump filters is somewhat trivial. But when you deal with fields that are less than a byte, things may get more difficult. That is why an entire section is devoted to writing tcpdump filters.

In case you are thinking to yourself that you'll never use tcpdump, so you'll never need to understand BPF theory, there are many applications that use BPF besides tcpdump. Snort, Scapy, Wireshark, and Tshark are a few of the many applications that can use BPF. Also, if you ever attempt to do programming with sockets or any of the other applications that can perform network programming, you may need to check a flag or value that requires some kind of bit selection or bit masking. So, this is quite a useful skill to apply to tools and tasks other than tcpdump.

Why Use Berkeley Filters

- BPF is supported by nearly *every* tool in this space.
 - Even when it's not exposed directly, the principles here apply in every tool.
- Any field in frame/packet
 - Examples: header length and TCP flags
- Macros for more commonly used fields:
 - Examples: tcp, port, and host
- Less common fields:
 - Identify protocol
 - Filter by values at any offset within the packet

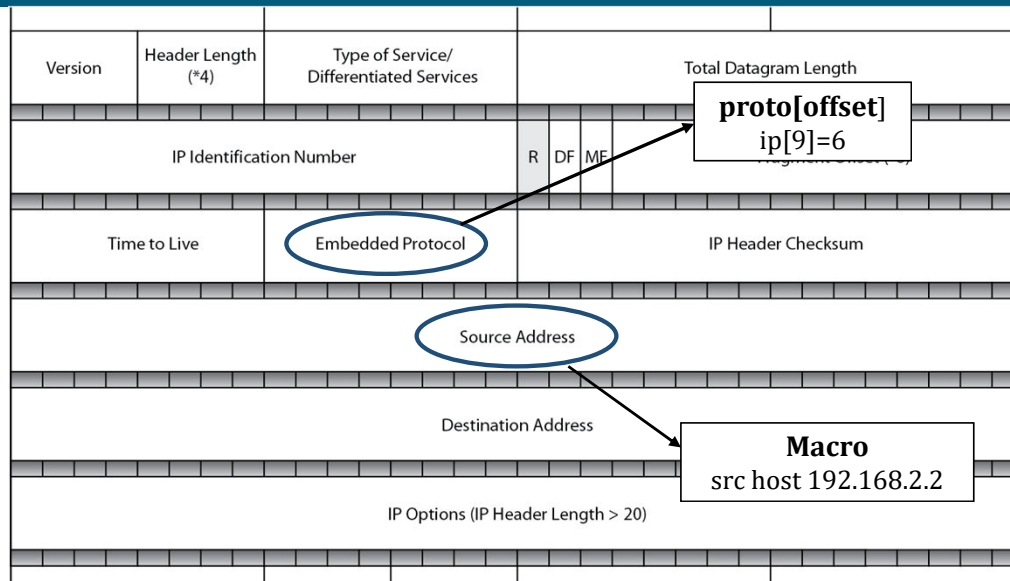
Tcpdump filters need to specify field(s) and value(s) of interest such as a bit, a byte, or multiple bytes found anywhere in the packet. The fields can be part of the IPv4 header (such as the IPv4 header length), the TCP header (such as TCP flags), the UDP header (such as the destination port), or the ICMP message (such as the message type). Even fields in the payload can be examined, though that is typically not done using BPF.

BPF has *macros* that can be used that represent common search criteria. They are really just handy keywords to use instead of describing the field in terms of the offset from the beginning of the protocol. For instance, the 'tcp' macro for IPv4 is really 'ip[9]=6'.

There are macros for the commonly used protocols, such as 'ip', 'tcp', 'udp', and 'icmp'. This is particularly helpful—even necessary—as it allowed us to identify the protocol in the IPv4 header as we did for 'ip[9]=6'. Otherwise, we would have to use the Ethernet header as the starting location and examine the 2 bytes that begin at offset 14. This field is the Ethernet type in a standard Ethernet header, and a value of 0x0800 means IPv4 follows. We would then have to find the IP protocol field location starting at the beginning of the Ethernet header. This would not work if VLAN tagging was placed in the Ethernet header.

The format of protocol [offset] was used in our example of 'ip[9]=6'. We will spend a great deal of time looking at the TCP flag field that is 13 bytes from the beginning of the TCP header and identified using 'tcp[13]'.

Specifying Fields



BPF allows you to specify filters using either a byte-offset notation or, for a subset of commonly used fields, a macro notation. The byte-offset notation requires that you specify a header that will be used as the starting point (offset zero), followed by an offset in square brackets. A macro allows you to specify a value more naturally (an IP address, for instance).

This slide depicts the use of a protocol and bytes displacement to identify the embedded protocol field as 'ip[9]' and the IP source address as 'src host'. The tcpdump man page is quite extensive, discussing different aspects of BPF.

Following are some of the common macros used in filters:

host: IP address of a host

net: designate a subnet such as 'net 192.168'

port: source or destination port

src host: source host

dst host: destination host

src net: source subnet

dst net: destination subnet

src port: source port

dst port: destination port

icmp: protocol field ip[9] has a value of 1.

tcp: protocol field ip[9] has a value of 6.

udp: protocol field ip[9] has a decimal value of 17.

The tcpdump Filter Format Examples

The two different formats for a tcpdump filter follow:

<protocol header> [offset: length] <relation> <value>

```
ip[9] = 0x01
tcp[2:2] != 80
udp[4:2] > 0
icmp[0] = 0x03 && icmp[1] = 0x01
```

<macro> <value>

```
port 22
dst host 192.168.11.62
src net 192
```

Let's look at examples of some other tcpdump filters. The top box uses the more primitive syntax of protocol and offset. The default is to examine the 1 byte at the offset location. You have to specify the number of bytes if it is more than one, using the syntax seen in the second example. It says to find the location 2 bytes from the beginning of the TCP header and grab 2 bytes. This is the location of the destination port; the filter selects records that are not port 80. You can examine up to 4 bytes using this format. Oddly, though, while you can filter on a single byte, 2 contiguous bytes, or 4 contiguous bytes, you cannot specify 3 contiguous bytes.

If you have a field that is greater than 4 contiguous bytes, you have to write a separate filter for each 4-byte range. Next, some kind of relation operator is used (for instance, "="). Finally, a comparison value must be given as well. It is easy to confuse BPF format with Wireshark display filter format. An equal sign in Wireshark is designated as ==. It just takes some time using both of these tools to get familiar with the syntax.

The second format uses a macro if one is available. It is a whole lot easier to use; you just specify the macro/variable name for the field you are examining and the value.

Let's look at the meaning of the filters in the slide:

The first filter, `ip[9] = 0x01`, selects any record with the IPv4 protocol of 1 (ICMP).

The second filter, `tcp[2:2] != 80`, selects any record where the destination port is not 80.

The third filter, `udp[4:2] > 0`, selects any UDP record with a nonzero UDP length, not a jumbogram.

The fourth filter, `icmp[0] = 0x03 && icmp[1] = 0x01`, selects any record with an ICMPv4 message type of 3 and a code of 1 (a "host unreachable" message).

The first macro filter selects any record with source/destination port of 22 (ssh).

The second macro filter selects any record with destination host 192.168.11.62.

The third macro filter selects any record with a source subnet of 192.x.x.x.

BPF Bit/Byte Refresher

Byte is an 8-bit field

- Possible to denote a span of bytes: `udp[0:2]`
- We can specify 1, 2, or 4 bytes.
 - You can make other lengths: `'udp[12:4]=0x0` and `udp[16:2]=0x02'`

Smallest precision BPF easily offers is a byte

- Many network, transport, and application protocols use fields that are smaller than a byte in length.
- These sub-byte values can be accessed with bit masking.

This is a refresher of bits and bytes from Section 1 to prepare you to write some filters. The bit is the smallest unit that can be represented by a computer; it can have a value of either 0 or 1. A byte is composed of 8 bits. Byte offset counting begins at byte 0; all successive bytes fall on these 8-bit boundaries.

The BPF language allows us to look at a single byte (by default) or a span of 2 or 4 bytes. To specify a span of bytes, we need to express which byte offset our first byte is found in and then indicate how many bytes should be extracted. For example, if we wanted to look at 4 bytes beginning at offset 12 from the start of the UDP header, we would express this as `'udp[12:4]'`.

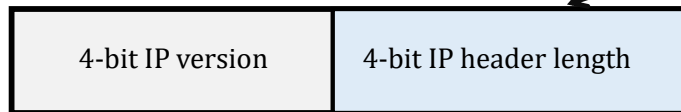
What if we need to examine 5 bytes, 6 bytes, or some other number of bytes other than 1, 2, or 4? We can simply combine expressions. For example, if we wanted to check if the 6 bytes beginning at offset 12 from the start of the UDP header contain `0x000000000002`, we might express this as:

```
'udp[12:4]=0x00 and udp[16:2]=0x02'
```

BPF format does not have an easy way to represent a single bit or multiple bits in a byte that you may want to examine. For instance, the 0-byte offset of the IP header has two fields, each a nibble long. If you want to look at the 4-bit version number or 4-bit IP header length, you will have to isolate those bits from the others in the byte. This is done using a combination of hex characters, the Boolean AND ("&") operator and, possibly, something known as bit masking.

The Problem: Looking at Fields Less Than a Byte Long

Layout of 0-offset byte of IPv4 header



Examine IP header length only.

What if you want to find all packets where the header length is greater than 20?

How about: **ip[0] > 0x45?**

Fails with a value of 0x55, 0x75, etc.

The problem is that the presence of the IP version value is making this difficult.

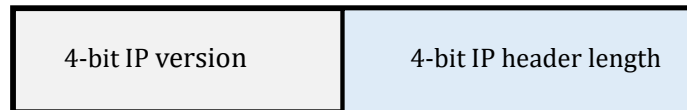
We run into difficulty writing filters when a field is less than a byte in length. For instance, suppose you want to find all packets where the IP header length is greater than 20. Remember that the IP header length needs to be multiplied by 4 to arrive at the actual value. A value of 5 in the IP header length signifies that the header is 20 bytes long. The 0-byte offset of the IP header actually contains two different fields: a 4-bit IP version and a 4-bit IP header length. If we use the **protocol[displacement]** notation, ip[0] finds both fields because it encompasses the entire byte.

We are not interested in the 4-bit IP version and essentially want to get rid of it somehow, or more correctly, we want to get rid of the value found in the IP version. We want to examine each packet for a value of greater than 5 in the IP header length field by isolating it from the IP version.

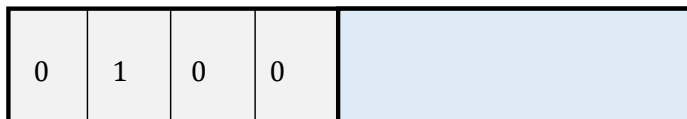
Let's make a first attempt using the filter 'ip[0] > 0x45'. On the surface, this looks like it should work. But suppose an invalid IP version number of greater than 5 is found, yet the header length value is 5. The filter will select that packet, thus not working as we intended. The presence of a value in the IP version complicates our filter. We want to exclude the IP version value from our comparison.

The Solution: Nullify the IP Version Value

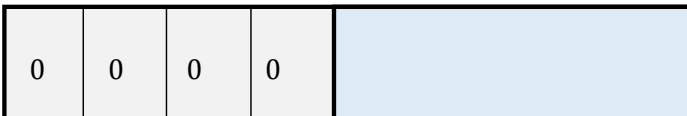
Layout of 0-offset byte



Current value in IP version



Desired value in IP version



As we discovered, there is no simple operation that is native to BPF that enables us to look exclusively at the IP header length value. But we can do some mathematical operations and manipulations of fields and bits that enable us to look at the 4-bit IP header length only.

In essence, if we can nullify all the bits in the IP version field to be 0, we can actually look at just the 4-bit IP header length when we look at `ip[0]`. How exactly do we discard or zero out this high-order nibble and preserve the low-order nibble found in the 4-bit header length?

More Fundamentals

Bits can be selected using bit masking.

- Uses the Boolean "&" (and) operation to keep or discard bits
- Ones preserve, zeros destroy
- When bits are ANDed, the following values yield the following results:

Truth Table		
&	True	False
True	True	False
False	False	False

Our Packet Bits		
&	1	0
1	1	0
0	0	0

We use the Boolean "&" operation to zero out unwanted bits. Let's look at the fundamentals of applying this theory. The "&" operator is used because it can easily discard unwanted bits by AND'ing a bit found in the packet with another bit of 0. This is known as the *mask bit*; we select the appropriate value (either 0 or 1) to discard or preserve each individual bit in the IP version and IP header length. The AND operation can preserve packet bit values by using a mask bit of 1. Similarly, a bit mask value of 0 nullifies the packet bit.

Take a closer look at the AND truth table in the slide. Because we deal with computers that talk in binary, we can consider taking every combination of the only two possible bit values: 0 and 1. As you can see from the truth table, the only time AND'ing 2 bits has a resulting value of 1 is when both bits are 1.

The process that we follow is to take a bit found in the packet byte (say, in the IP header length field) and then select an appropriate mask bit for the AND operation. The appropriate mask bit is one that either preserves the packet bit when the mask bit value is 1 or discards the packet bit when the mask bit value is 0.

Solution: "AND" Unwanted Bits with 0s

Current value in IP version

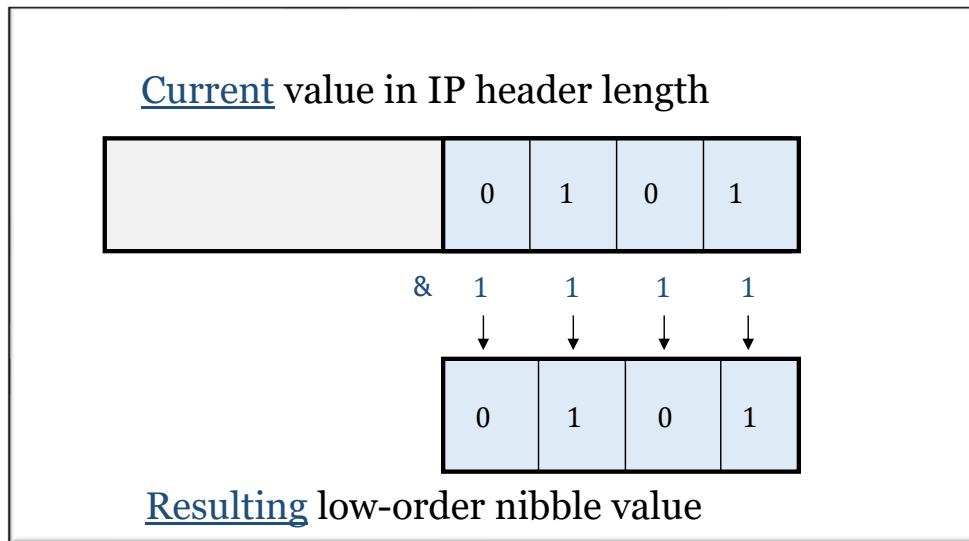
0	1	0	0	
---	---	---	---	--

&

0	0	0	0
↓	↓	↓	↓
0	0	0	0

Resulting high-order nibble value

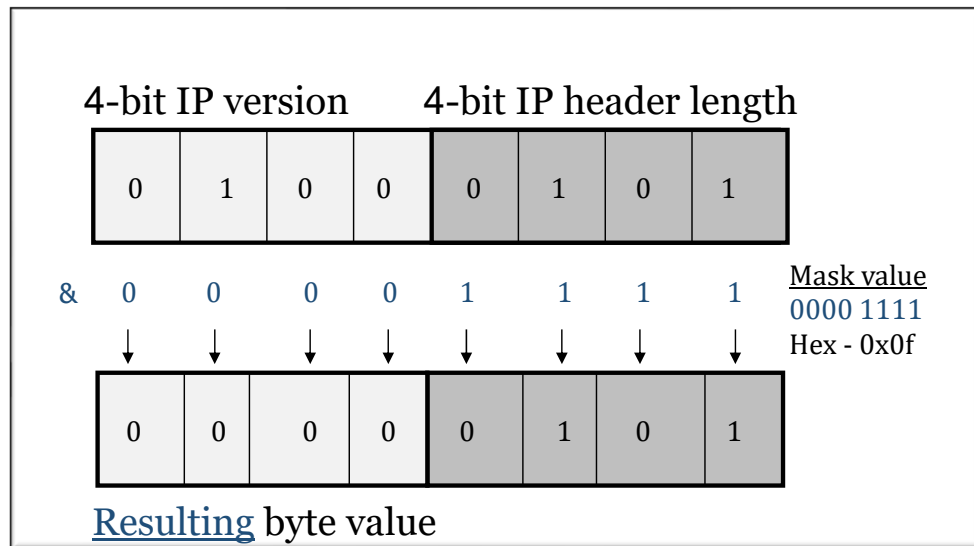
A solution for dealing with fields that are less than a byte is basically to zero out all other bits in the byte other than those we want to examine. In this instance, we want to AND the high-order 4 bits in the 0-byte offset of the IP header with zeros. This operation yields a result where there are zeros in the place where there once might have been nonzero values. Therefore, we have a high-order nibble mask with a value of 0000.

Solution: "AND" Wanted Bits with 1s

Because we deal with an entire byte, we must also pay attention to the low-order nibble. This contains the IP header length that we want to preserve. We must preserve the packet bit values that we found there.

To preserve the current value found in that field, we use the AND operator with a nibble with all the bits turned on or set to '1'. In other words, we have a mask with a binary value of 1111 for this nibble.

The Mask Byte



Ultimately, what you have to do is create an appropriate mask byte. This is a byte that will be AND'ed with the packet value found in the 0-byte offset of the IP header to give you the wanted resulting byte, which has the high-order nibble of all zeros and the low-order nibble unchanged. So, this just means that your mask byte is a binary 0000 1111, which is translated to two hexadecimal characters of 0x0f.

You may be wondering why we are using hexadecimal instead of decimal. Decimal values work well in daily life, but bits and bytes are more easily translated into hexadecimal. We will express mask and comparison values in hexadecimal, which will become more intuitive, as we hope you soon will see.

It is important to note that you must specify a prefix of 0x to denote hexadecimal. Otherwise, BPF thinks that you mean decimal. You would think it could recognize the value of "0f" as hexadecimal, but it returns a "syntax error."

Begin Putting It All Together

Current value in first byte of IP header

0	1	0	0	0	1	0	1
---	---	---	---	---	---	---	---

0 0 0 0 1 1 1 1

Mask value
0000 1111
Hex - 0x0f

Partial filter = `ip[0] & 0x0f`

↓ ↓ ↓
field AND mask

We have discovered that the mask byte we need is 0x0f. We need to denote the 0-byte offset of the IP header and AND it with the mask byte. But how do we tell BPF to do this? First, identify the byte (or bytes) of interest by identifying the protocol and byte displacement. Next, use the "&" symbol to denote the Boolean AND operation and complete the operation using the mask value. The expression so far is 'ip[0] & 0x0f'.

An important point is that you must designate bit masks that are minimally a byte in length. Sure, you may be dealing with a single bit or more in the particular byte, but the byte is the smallest unit used to specify a mask. Also, be aware that tcpdump filter syntax allows the use of "&" as well as "and," but in two entirely different contexts. The "&" is used exclusively in bit masking, whereas "and" is used to concatenate two different criteria in a single filter (for instance, 'tcp port 80 and host 192.168.1.2').

You must understand that using a filter of any kind, mask or no mask, does not alter the value in the packets. It simply specifies the location and values used to analyze the packets.

Bit Masking: TCP Flag Byte

- TCP flag field located in the TCP header
- Describes the state of a given TCP segment

2^3	2^2	2^1	2^0	2^3	2^2	2^1	2^0
CWR	ECE	URG	ACK	PSH	RST	SYN	FIN

This field is denoted as `tcp[13]`.

Don't worry about the exact meanings of these yet—we will cover them in detail later.

There will be many times you want to test a TCP segment for a certain TCP flag setting or multiple TCP flag settings. Again, we revisit the problem of trying to look at a bit or several bits in a byte.

Why might you want to look at particular bits only? Say you are interested in viewing only TCP segments with either the SYN bit only set or SYN and ACK flags set together. This can tell you whether a connection was attempted and established. Perhaps you want to find all TCP sessions that were aborted with an RST.

Is a Mask Byte Always Needed?

2^3	2^2	2^1	2^0	2^3	2^2	2^1	2^0
CWR	ECE	URG	ACK	PSH	RST	SYN	FIN

- Not all bit selections/filters require a mask byte.
- When specific bits only must be set, yet all others must not be set, a mask is not required.
- This condition may be tested with an exact value:
 - SYN bit only set `tcp[13] = 0x02`
 - ACK and PUSH only set `tcp[13] = 0x18`
- If there is uncertainty whether other bits in the byte may be set, a mask byte is required.

A mask may not be required when dealing with one or more bits in a given byte of interest. If the condition you designate is an exact value, a mask is not required. This means you know that one or more bits must be set, and no other bit may be set. You can specify this condition as you would any other filter.

Say that you want to find all packets where the SYN flag alone is set. You know that the SYN flag is in the 2^1 position in the low-order nibble; therefore, you can specify its value as 0x02. Perhaps you would like to filter packets where both the ACK and PUSH flags only are set. Again, an exact value of 0x18 specifies this condition.

Going back to the original example, where we wanted to test or evaluate the IP header length field, we used a mask to discard the high-order nibble of the byte and preserve the low-order nibble of the 0-offset byte. Why was this necessary if we wanted to test for an IPv4 header length of 5? The high-order nibble is the IPv4 version number. We don't know if this is 4 or if it is 6 or any other value. And, frankly, we do not care about this value if our concern is the IPv4 header length. In this case, we needed to discard the high-order nibble and preserve the low-order nibble. The only way to specify that filter is by using a mask byte.

Section 14.2 Packet Protocols as a Second Language	Section 14.3 Scapy	Section 14.4 Zeek	Section 14.5 Wireshark	Section 14.6 tcpdump
- 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 - Scapy & 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-up expert analysts' progression through real-world data

- Wireshark Display Filters
- Writing tcpdump Filters
- TCP
- UDP
- ICMP
- IPv6

Fundamentals of Traffic Analysis: Part II

Understanding tcpdump Filters

Three Categories of Filters

Miscellaneous Filter Topics

This page intentionally left blank.

Three Categories of Filters

- Most exclusive: All designated bit(s) must be set; no other bits can be set = exact value.
- Less exclusive: All designated bit(s) must be set; other bits can be set.
- Least exclusive: At least one of the designated bits must be set; other bits can be set.

For our purposes, we categorize the types of filters we use. We will refer to the first as the most exclusive. This is where all designated bits must be set, yet no other bits can be set. For instance, perhaps we want to determine whether both the SYN and FIN bits only are set.

The second category is the less exclusive one. Continuing with the same example, this would be where both the SYN and FIN bits are set, yet any of the other flag bits may be set, too.

The final category is the least exclusive one. This is where either the SYN or FIN must be set, yet any of the other flag bits may be set, too.

Find SYN, FIN: Most Exclusive

192.168.1.103.1030 > 192.168.1.104.80: Flags [SF], seq 0, win 8192, length 0

Select any packet that has **both** SYN and FIN set, but **no other** flag bits are set.

 AND

2 ³	2 ²	2 ¹	2 ⁰	2 ³	2 ²	2 ¹	2 ⁰
CWR	ECE	URG	ACK	PSH	RST	SYN	FIN
0	0	0	0	0	0	1	1

Are the bits *exactly* this?

```
tcpdump -nt -r bpf-filter.pcap 'tcp[13] = 0x03'
```



Let's go through some examples of the filter categories.

Apply the filter of 'tcp[13] = 0x03'. The only combination of bit settings that equal 0x03 is when the SYN and FIN bits alone are set. These are the only bits that can be set. Any other bits that are set cause the packet to have a different TCP flag byte value than 0x03.

The file **bpf-filter.pcap** is used as input for tcpdump. The output displayed from this filter is the one packet with only the SYN and FIN bits set. Remember that bit masking is not required when there is an exact known value for the byte. This can be considered the most exclusive type of filter, to be used when dealing with a byte with distinct and known bit fields and values.



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

```
tcpdump -nt -r bpf-filter.pcap 'tcp[13] = 0x03'
```

Find SYN, FIN: Less Exclusive

192.168.1.103.1030 > 192.168.1.104.80: Flags [FS]
 192.168.1.103.1030 > 192.168.1.104.80: Flags [FSRP.UEW], ack 0

Select any packet that has both SYN and FIN set and can have any other flag bits set.



2 ³	2 ²	2 ¹	2 ⁰	2 ³	2 ²	2 ¹	2 ⁰
CWR	ECE	URG	ACK	PSH	RST	SYN	FIN
0	0	0	0	0	0	1	1

Are both set?

```
tcpdump -nt -r bpf-filter.pcap 'tcp[13]&0x03 = 0x03'
```

Check these; ignore all others.

SANS



bpf-filter.pcap

SEC503: Intrusion Detection In-Depth 49

Let's examine a second filter that is less exclusive. Say we want to find all packets that have the SYN and FIN flags set, but we don't care if any other flag bits are set. We need a way to exclude all other flag bit settings. We need a mask byte in this case because an exact byte value cannot be determined; any other flag bit may be set.

The filter 'tcp[13] & 0x03 = 0x03' indicates using an "AND" operation with a mask byte of 0x03 on the packet's TCP flag byte. The result of this operation must be 0x03. Essentially, using a mask of 0x03 causes every bit (except the SYN and FIN) in the resulting flag byte to be 0 by AND'ing it with 0. We've focused the filter on the 2⁰ and 2¹ bits only.



Once you zero out the bits that may or may not be set, you are left with the same logic as the previous, most exclusive filter. You want the remaining bits to have a value of 0x03. That is possible only if the SYN and FIN flags are both set.

The tcpdump output reflects the resulting selected records when this filter is applied to **bpf-filter.pcap**. The first packet has the SYN and FIN flags alone set. The second displayed packet has the SYN and FIN flags set and every other one as well.

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

```
tcpdump -nt -r bpf-filter.pcap 'tcp[13] & 0x03 = 0x03'
```

Find SYN, FIN: Least Exclusive

```
192.168.1.103.1030 > 192.168.1.104.80: Flags [FS]
192.168.1.103.1030 > 192.168.1.104.80: Flags [FSRP.UEW], ack 0
192.168.1.103.1030 > 192.168.1.104.80: Flags [F]
192.168.1.103.1030 > 192.168.1.104.80: Flags [FR]
```

Select packets where SYN, FIN, or both are set.



2 ³	2 ²	2 ¹	2 ⁰	2 ³	2 ²	2 ¹	2 ⁰
CWR	ECE	URG	ACK	PSH	RST	SYN	FIN
0	0	0	0	0	0	1	1

Are either set?

```
tcpdump -nt -r bpf-filter.pcap 'tcp[13]&0x03 != 0'
```

After masking, is there anything left?

SANS



bpf-filter.pcap

SEC503: Intrusion Detection In-Depth 50

Finally, let's ease up even more by using a filter of 'tcp[13] & 0x03 !=0'. This filter indicates that the result of masking the TCP flag byte with a value of 0x03 must not be 0; in other words, it must be greater than 0. Again, using a mask of 0x03 causes every other bit—except the SYN and FIN—in the resulting flag byte to be masked with a 0.

We've isolated the SYN and FIN flags again. Consider the following: If the SYN bit alone is set, the resulting flag byte value is 0x02; if the FIN bit alone is set, the value is 0x01; and if both bits are set, the value is 0x03. What is the only condition where none of these three cases is true? That would mean that neither the SYN nor the FIN bit is set. The value of the byte would be 0x00. Following this logic, having either or both of the SYN/FIN bits set results in a nonzero value. That is why the logic of "!=0" is used after applying the proper mask.



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

```
tcpdump -nt -r bpf-filter.pcap 'tcp[13] & 0x03 !=0'
```

A Couple More Practice Cases

2 ³	2 ²	2 ¹	2 ⁰	2 ³	2 ²	2 ¹	2 ⁰
CWR	ECE	URG	ACK	PSH	RST	SYN	FIN

- 1) What would the filter be to check that **both** the **FIN** and **RST** bits are set (no other flag bits are set)?
- 2) How would you find packets where **PUSH** and **ACK** are both set and other flags can be set?
- 3) What would the mask be to test if **either** or **both** the **PUSH** and/or **FIN** flags are set? (Other flags may be set, too.) What would the filter be for this?

Here is a worksheet for figuring out these filters:

Filter 1 = _____ [_____] = 0x _____
protocol [bytes]

Filter 2 = _____ [_____] & 0x _____ != 0
protocol [bytes]

Filter 3 = _____ [_____] & 0x _____ != 0
protocol [bytes]

Answers:

Filter 1: tcp[13] = 0x05

Filter 2: tcp[13] & 0x18 != 0

Filter 3: tcp[13] & 0x09 != 0

Section 14.2
Packet Protocols as
a Second Language

- An immersion class in speaking packet protocols, and hex
- Ethernet
- IPv4 & IPv6
- TCP, UDP, & ICMP
- Wireshark & tcpdump fundamentals, filtering, & investigation

- Snaps 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 fellow expert analysts' progression through real-world data

Prerequisite: Section 14.1: Packet Protocols as a Second Language

Fundamentals of Traffic Analysis: Part II

Understanding tcpdump Filters

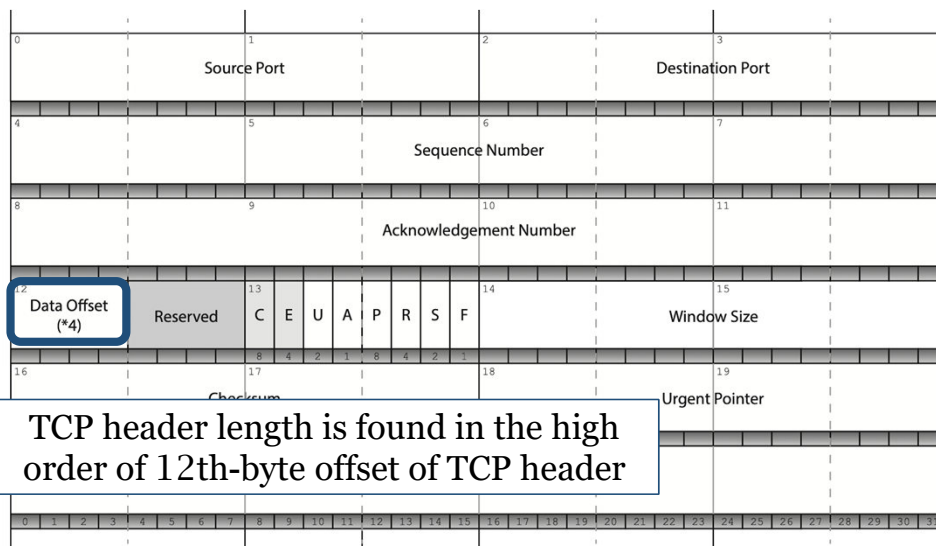
Three Categories of Filters

Miscellaneous Filter Topics

- Wireshark Display Filters
- Writing tcpdump Filters
- TCP
- UDP
- ICMP
- IPv6

This page intentionally left blank.

Extract TCP Header Length



Let's take a look at the situation where you would like to look at the TCP header length, perhaps to see if there are TCP options. We have not covered the TCP header length in any detail yet, but like the IP header length, it is a 4-bit field that must be multiplied by 4 to represent the true header length. A standard TCP header with no TCP options is 20 bytes. However, unlike IP, TCP options are common and often used, especially on any segment where the SYN flag is set.

The TCP header length is located in the high-order nibble of the 12th-byte offset from the beginning of the TCP header. Therefore, if we want to extract the value for comparison, we must first discard the low-order nibble.

In addition to using masking to discard the low-order nibble of the byte, the location in the high-order nibble causes the value to be 16 times more than if it were in the low-order nibble. Remember, each 4-bit nibble represents an incrementing power of 16. When you're dealing with a single byte, the lower-order nibble is 16^0 and the high-order nibble is 16^1 .

A value of 0x50 in this byte means it is 80 decimal. This is not a problem per se; you just need to take this into consideration when you identify a comparison value.

A Potpourri of Options: Find TCP Header Length Greater Than 5

Option 1: Discard low-order nibble and compare high-order nibble value:

`'tcp[12] & 0xf0 > 0x50'`

Option 2: Divide by 16 to treat it as a low-order nibble value:

`'tcp[12]/16 > 5'`

(alternatively, dividing by a power of 2 is also known as a "right shift"):

`'tcp[12] >> 4 > 5'`

There are several ways to deal with the issue of a value in the high-order nibble. But before you investigate your options, whenever you have a value in the high-order nibble, remember that you need to manipulate that value. In other words, you cannot just compare the value to be greater than 5. You always need to consider any value you find in any byte in terms of its value in the byte, not just the bits that it occupies. This seems easy to grasp right now as we cover it, but the concept may disappear as soon as you flip to the next page, given the deluge of new information you are assimilating.

The first option is to leave the high-order nibble value as it is, use bit masking to nullify the low-order nibble, and compare the remaining value with 0x50. This option expresses the comparison value in terms of being located in the high-order nibble as well.

A second option is to divide the high-order nibble value by 16. Another way to look at the division is as a "right shift." Whenever you divide a binary number by a power of two, you are effectively shifting all of the bits to the right by one place. If you are dividing by 16, that is the same as shifting to the right 4 places, since 2^4 is 16.

Option 1: Mask and Alter Comparison Value

Discard low-order nibble and compare with high-order nibble value intact:

```
'tcp[12] & 0xf0 > 0x50'
```

	Preserve				Discard				
	0	1	0	1	?	?	?	?	
Mask	1	1	1	1	0	0	0	0	
	0	1	0	1	0	0	0	0	= 0x50

Option 1 is nothing new in regard to discarding unwanted bits. It is what we've been doing all along; however, now our attention is on the high-order nibble. We nullify the low-order nibble by using a bit mask of 0xf0 and performing a binary AND operation on it.

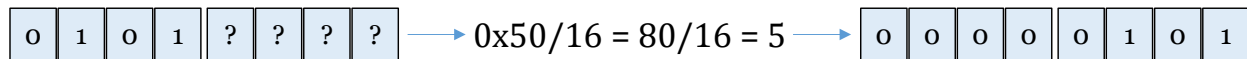
We are left with the high-order nibble value preserved. We can choose to simply compare it against a value represented in the high-order nibble (in this case, 0x50).

Option 2: Divide by 16

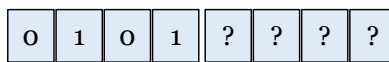
To discard low-order nibble, divide by 16:

```
'tcp[12]/16 > 0x05'
```

Move these bits



```
'tcp[12] >> 4 > 0x05'
```



Bit Bucket

Our second option lets us deal with the high-order nibble value as if it were in the low-order nibble. As you know, the high-order nibble falls in the 16^1 position, while the low-order nibble falls in the 16^0 position. Therefore, the high-order nibble value is 16 times greater than if it were in the low-order nibble.

We can divide by 16, essentially making it a low-order nibble value. BPF supports some basic math functions, such as division, multiplication, addition, and subtraction. As we've seen, it also supports bitwise AND operations, as well as bitwise OR. It can also shift bits right or left, as we'll see in the next slide.

One thing to remember about BPF is that it is a low-level language that does primitive operations. However, because it is such a low-level language, it is far more efficient than all the abstraction and processing required by Wireshark display filters.

An alternate way to accomplish this same effect is to discard the low-order nibble value by shifting it right 4 bits. A shift operation moves the bits right or left. The ">>" denotes move to the right, and "<<" denotes move to the left. You need to designate the number of bits to shift. In this case, we simply shift the values 4 bits to the right. This moves the low-order nibble values into "bit oblivion," if you will. They just shift off the bit cliff. This positions the high-order nibble value in the lower order, just where we want it.

Bit shifting either divides (right shift) or multiplies (left shift) the existing value by a power of 2^n , where n is the number of shift bits specified. In this case, we divided by 16 in a different way.

Wrap-Up of Writing tcpdump Filters

- Use of macros or offset/number of bytes to inspect
- Examining fields less than a byte long may require the use of masking and comparison operations and values.
- Filters can be more or less exclusive as required.

In conclusion, the use of BPF is straightforward when you examine a field or fields that fall on byte boundaries. There are some macros that can be used to identify commonly used fields. If no macro is available, you reference a field by its protocol and displacement in bytes from the beginning of the protocol header, and optionally the number of bytes to examine if more than a single byte. Things get a bit more complex when you examine fields that do not fall on byte boundaries. You may have to use bit masking to selectively preserve and discard packet bit values to yield the wanted result.

There are ways to make filters more or less exclusive. It is possible to filter most exclusively when there are known exact bit settings/values. This requires no mask. You can filter less exclusively by applying a mask to discard any bit settings/values that are irrelevant, the result yielding a specific byte value. Finally, it is possible to filter least exclusively by applying a mask to discard any bit settings/values that are irrelevant with a resulting nonzero value. This permits you to designate a choice of bit settings/values.

Writing tcpdump Filters Exercises

Workbook exercises "Writing tcpdump Filters"

This page intentionally left blank.

Session 14.2 Packet Protocols as a Second Language

• An immersion class in speaking packet 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

© 2021 SANS Institute. All rights reserved. No part of this publication may be reproduced without prior written permission from SANS Institute.

- Wireshark Display Filters
- Writing tcpdump Filters
- TCP
- UDP
- ICMP
- IPv6

Fundamentals of Traffic Analysis: Part II

TCP Theory

TCP Header Fields

TCP Stimulus and Response

We've finally arrived at the transport layer. The first of three protocols that we examine is *Transmission Control Protocol (TCP)*, which is a complex protocol with many concerns for analysts. It's important to understand how to dissect packets that use TCP as the transport protocol, and to understand TCP theory, operation, and fields associated with TCP. TCP is a reliable protocol, unlike IP, UDP, and ICMP. As such, it requires a means to determine whether packets have been received.

Also, TCP has a true notion of a session. One of the characteristics of a session is that there are multiple individual packets containing TCP *segments* (the term used for a TCP header and data) that comprise a single session. These segments are associated with each other and must have a way to indicate chronology. There are standards for how a TCP session begins, how data is exchanged, and how the session closes.

You need to become familiar with all these concepts and more to understand normal TCP behavior and potential nonstandard behavior.

TCP

- Supports unicast addresses only
- Provides reliability
- Sequences data for ordering purposes
- Supplies flow control to optimize efficient data exchange
- Accounts for more than 90% of internet traffic
- Encapsulated in IP header (protocol/next header value 6)

As we'll see, TCP has to synchronize a connection between hosts. Because of this, it is only possible for one source host to talk to one destination host. This is known as a *unicast* connection. TCP provides for reliable delivery. This comes in the form of a sending host designating a sequence number for each TCP segment and the destination host acknowledging all received data. This is the only way that the sender knows that the data has been received.

The TCP sequence number in the TCP header represents a chronological value assigned to the first byte of data in each TCP segment sent. Because TCP segments are transported in IP packets and because IP packets can take different routes to the destination host, it is possible for data to arrive in a different order than it was sent. Therefore, the receiving host's TCP is responsible for reordering the data by using the TCP sequence numbers.

TCP attempts to optimize the flow of data between the hosts to ensure that as much data as possible can be exchanged without overloading the network or the receiving host. Approximately 90% or more of the traffic on the internet is TCP, so you can see that most data exchanges demand reliability. Finally, don't forget that the TCP header and data are encapsulated in an IP header with a protocol/next header value of 6.

What Is Reliability?

- Reliability from a networking perspective:
 - This isn't questioning whether packets will arrive!
 - If your network is "unreliable," you should fix it.
 - A protocol's ability to detect and correct or address communication errors *during* the connection
- TCP:
 - Detects missing, out-of-order, and desynchronization problems
 - Has mechanisms that seek to correct these situations
- UDP:
 - Has no means of detecting out-of-order, missing, or other problems *during* the "connection"

Before we go on, the matter of "reliability" deserves some additional discussion. Sometimes students may think that "reliable" versus "unreliable" is stating that the protocol is somehow inherently prone to losing information. This is not the case! When it comes to network protocols, the word "reliability" has a very specific meaning.

Specifically, reliability refers to built-in mechanisms within a protocol that can detect problems *during* the data transmission (as opposed to during session or connection initiation; all protocols have support to detect problems here) and act to correct or otherwise resolve those problems. For example, during a TCP conversation, an endpoint, by virtue of the protocol, will immediately know if data has been missed. As soon as a host detects that data has been missed, it will immediately request that the missing data be retransmitted, thus both detecting and acting to correct the problem. In fact, for a programmer, TCP is an extremely attractive option.

The programmer simply needs to tell the network library that she wants to connect to a host on a particular port. As long as that succeeds, the programmer can stuff data into the socket and assume (rightly) that as long as no error is reported, that data has been sent and received by the other computer. Obviously, there is a significant amount of overhead that comes with this. This means that, while TCP is much easier for a programmer to use, UDP having less overhead would actually be much faster in the long run. Of course, if UDP were used, all the reliability would become the responsibility of the programmer.

Thinking About TCP

Stop thinking about TCP as a single full-duplex connection.

- Start thinking about it as two separate, independent, half-duplex connections:
 - Host A connects to Host B from a source port to a destination port.
 - Host B connects to Host A from a source port to a destination port.

TCP is *all about* fragmentation.

- When you send a file over TCP, it is broken into many pieces *by the protocol*.
- We just call it *segmentation*.

Thinking in these ways helps explain some behaviors.

Before we go on, we'd like to encourage you to think about TCP in a way that you likely have never thought about it before. TCP is almost always described as establishing a "full-duplex" connection, which means that data can travel in both directions in the connection. This is not strictly true.

In fact, when a TCP session is initiated, one host establishes a connection to the other host. This creates a half-duplex channel, allowing data to pass in one direction. It just so happens that the other host will simultaneously establish a connection back to the first host. This second connection, also half-duplex, is what allows data to flow in the other direction.

This might seem like a really fine point. However, remember that the approach we are taking is a bottom-up discussion of the protocols. The hope is that this will give you an intimate understanding of how the protocols work, which will be tremendously helpful when either troubleshooting or attempting to figure out what an attacker is trying to do (and why it works). If you think about the three-way handshake, we can actually break it down slightly differently.

Host A sends a SYN to Host B. Host B responds with an ACK, accepting the connection. As a result of Host A's connection, Host B decides to connect to Host A, so it sends a SYN to Host A. Host A, accepting that connection, responds with an ACK.

In practice, this is simplified, collapsing into the SYN -> SYN/ACK -> ACK. However, hopefully you have a better understanding of *why* a three-way handshake is used.

It is also useful to realize that TCP is really all about fragmentation. We have some connection of arbitrary length that is used to carry a conversation. That conversation is broken into many pieces. Under IP, when we break things into pieces, we call them *fragments*. Here, we call these pieces *segments*.

Session 14: 2 Packet/Network as a Second Language	Session 15: 2 Scapy for building tools and simulating signatures & behaviors	Session 16: 2 Architecting for massive packet capture & analysis	Session 17: 2 Guided scenarios to reinforce all skills	Session 18: 2 Challenge questions guide/follow expert analysts' progression through real-world data
- 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 - Scout & 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

- Wireshark Display Filters
- Writing tcpdump Filters
- TCP
- UDP
- ICMP
- IPv6

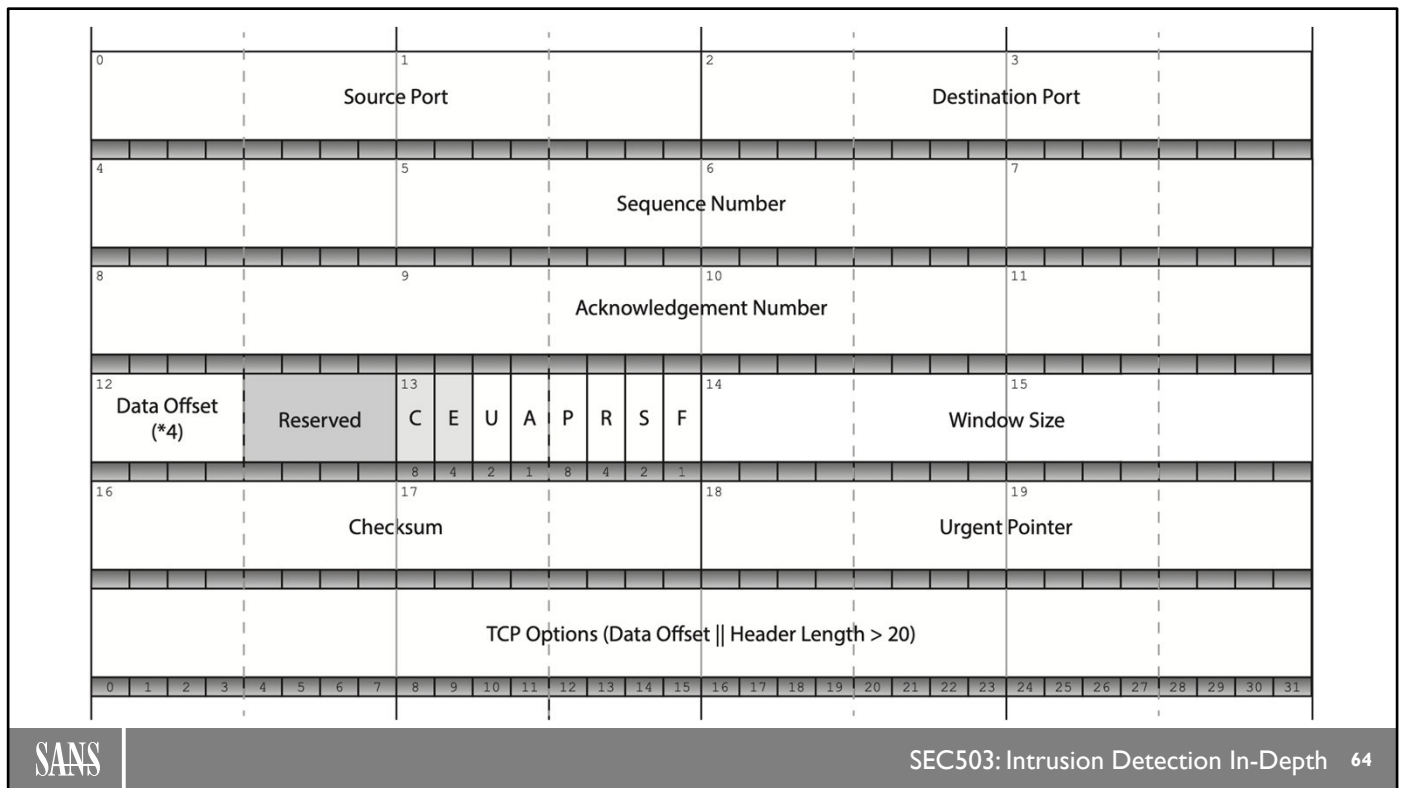
Fundamentals of Traffic Analysis: Part II

TCP Theory

TCP Header Fields

TCP Stimulus and Response

With the theory out of the way, we're ready to march through the different header fields. Be prepared; this is a very long section with many details. Don't hesitate to take a break when you need it. If you're doing this self-study or OnDemand, you can also take a break by working on some TCP labs to break things up. Whenever you get to something in the lab that's unfamiliar, go back to the book and pick up studying again!



Pictured above is the TCP header. Even though TCP is a very familiar protocol, there will likely be a number of things about TCP of which you were unaware or for which you had an incomplete (or even inaccurate) understanding. Since TCP is so widely used, it is important that your understanding of it be as complete and accurate as possible. For this reason, the TCP material will occupy the greatest amount of our attention in this volume.

As with the IP material in the first book, you will see a small TCP header that acts as a legend to help to orient you during our discussion. We will work our way through the entire header, addressing each field's purpose, common behaviors, and normal values.

TCP Ports



16-bit fields representing source and destination ports

- Valid values 0–65535 - 0 is valid; however, there is no socket support to listen on it.

Source port typically ephemeral (temporary)

- 1024–65535
- Different OSs use different ranges!
- Source port changes for each new connection

Linux	32768–61000
BSD <4.6	1024–5000
Windows XP	1025–5000
Server 2008	49152–65535
2008 + Exchange	1025–60000

Destination port must be known

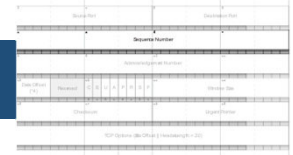
- Typically < 1024, but not a requirement
- IANA maintains well-known list, but any service can run on any port.

The port fields are two separate 16-bit fields in the TCP header: one for source and another for destination ports. The range of values is between 0 and 65535. A value of 0 as a TCP port number is an acceptable value to be placed in the header; however, in a conventional TCP/IP stack, there is no support for the use of port 0, as there is no means to open a socket on port 0. This is because port 0 is used in the network programming library to request an ephemeral, or temporary, port. This means 0 is never used as a port in traffic.

When a source host wants to connect to a destination host, an ephemeral port is typically selected in the range of ports greater than 1023, though the actual range used by an operating system is operating system dependent (and typically configurable). IANA recommends the port range 49152 through 65535 be used as ephemeral ports, but you will find that many operating systems ignore this advice.

For each new sent connection that the source host attempts, a different ephemeral port should be selected. The ephemeral ports typically increment for every new connection attempt, though you will find variation. For example, BSD systems may use a random ephemeral port for each new connection. In a scan scenario, you will often see the source port value incrementing by 1 for each new connection when the scanner is attempting to *actually* connect. When the scanner is simply probing ports, you will often see the source port remain static across multiple different apparent connection attempts.

TCP Sequence Numbers



TCP data exchange done in streams of multiple segments

- Used to order each segment sent
- 32-bit number is the offset from ISN of the initial byte in the segment

Should change for each new (non-retry) TCP segment sent

- Not "new packet," "new segment" – What's the difference?

Initial sequence number (ISN) represents zero offset

Value increments by number of bytes in payload

- ACK in response = SEQ + # Bytes Sent
- By RFC, SYN, and FIN count as 1 byte in the connection

When a data transfer occurs over TCP, it is likely that it will have to be done using many different packets, each with its own TCP *segment*. As noted, TCP is a reliable protocol. As such, it must know when packets have not been delivered. It also needs to correctly reorder the packets when they arrive since we have no control over the path that each packet might take across the network. This means that packets may not arrive in the order that they were sent.

Each TCP segment that is sent has a TCP sequence number found in the TCP header. This is a 32-bit number that represents the first byte of data for the TCP segment. When a TCP session is started, each side of the session selects an ISN as the first sequence number. You can (and should) think of this ISN as the beginning *offset* of the data, which means that the ISN is equivalent to the zero-offset byte in the TCP connection.

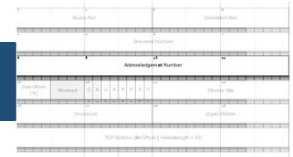
These TCP sequence numbers give the entire stream associated with the TCP exchange a notion of order. A TCP header contains a sequence number, allowing a packet that is lost to be identified. Also, if TCP segments arrive at the destination host in a different order than they were sent, the receiving host can correctly order them using the TCP sequence number.

A TCP sequence number is consumed with each byte of payload data. For instance, if the payload contains 1000 bytes of data, the TCP sequence number will increment by 1000. To be clear, the sequence number does not increase in *this* packet, but it will be 1000 greater in the *next* TCP packet that this host sends.

Another (accurate) way to think about this is that the sequence number is being used to track how many bytes have passed in a connection. In fact, not only is this how the data is "sequenced," or put in order, but this is also how an endpoint knows when data is missing.

To determine how much data has passed in a direction, you can simply subtract the current sequence number from the initial sequence number for the connection.

TCP Acknowledgement Numbers



TCP is a reliable protocol; guaranteed delivery

- Acknowledgement by receiver identifies receipt of sent data
- Acknowledgement number indicates the byte offset of the last byte received
- Remember, ISN is used to determine the “zero-offset” for the connection.

The SYN and FIN count as a byte.

Because TCP guarantees reliable delivery of all data, it needs a mechanism to detect whether TCP data has been received. The way the host knows that data has arrived is that the size of the packet is greater than the sum of the IP and TCP headers. When responding, the host must acknowledge receipt of data. It does so by first setting the acknowledgement flag in the TCP header. It is not enough to simply set the ACK bit, however. Recall that this bit simply means that the number in the acknowledgement field is valid. Therefore, the host must ensure that the number is indeed valid.

The way that it determines the correct number to insert is by taking the sequence number that is in the received packet, calculating the amount of data sent in the packet ($\text{IP total length} - (\text{IP header length} + \text{TCP header length})$), and adding this amount to the sequence number. This should be the correct acknowledgement number, indicating that this number of bytes was seen by the receiver.

If a packet is lost from receiver to sender, the receiving host gets a TCP sequence number greater than the one it expects. It knows that this is wrong; however, it is limited in the way it can respond. It can only relate back to the sending host that it is expecting the TCP sequence number associated with the lost packet. It may take several of these duplicate acknowledgement numbers of previously accepted data returned to the sender for the sender to become aware that it needs to retransmit the lost packet.

3whs TCP Sequence/Acknowledgment Numbers

```
sans@sec503: tcpdump -r 3whs.pcap -c 3 -ntS
```

```
IP 192.168.11.62.36609 > 192.168.11.1.80: Flags [S], seq 1407137694, win 5840, options [mss 1460, sackOK, TS val 169259530 ecr 0, nop, wscale 6], length 0
```

+1

```
IP 192.168.11.1.80 > 192.168.11.62.36609: Flags [S.], seq 743334558, ack 1407137695, win 5792, options [mss 1460, sackOK, TS val 414457892 ecr 169259530, nop, wscale 1], length 0
```

+1

```
IP 192.168.11.62.36609 > 192.168.11.1.80: Flags [.], seq 1407137695, ack 743334559, win 92, options [nop, nop, TS val 169259531 ecr 414457892], length 0
```

Edited for clarity

SANS



3whs.pcap

SEC503: Intrusion Detection In-Depth 68

Now, let's reexamine these same three records, but with a focus on TCP acknowledgement numbers. The acknowledgement number is processed only when the ACK flag is set; otherwise, the value is ignored. We'll talk more about TCP flags in an upcoming slide.

As TCP sequence numbers assign a chronological 32-bit number to a sender's TCP segments, the receiver returns an acknowledgement number indicating whether the packet was received. Upon receipt of the new chronological sequence number from the sender, the receiver returns an acknowledgement number that specifies the next expected sequence number from the sender. This is sometimes called an *expectational acknowledgement number*. The sender receives the acknowledgement and knows whether a segment arrived.

The second packet above reflects the server's receipt of the sender's SYN/ACK segment that carried a TCP sequence number of 1407137694. The server issues an acknowledgement number of 1407137695. That is the sequence number that the server next expects from the client. The third record reflects the client's acknowledgement value of 743334559 of the server's SYN segment that carried an ISN of 743334558.



To follow along, enter the following command:

```
tcpdump -r 3whs.pcap -c 3 -ntS
```

Relative Sequence/Acknowledgement Numbers

Relative (Default display)

```
192.168.2.43.56380 > 172.217.10.129.443: Flags [S], seq 1323118408
172.217.10.129.443 > 192.168.2.43.56380: Flags [S.], seq 3183094287, ack
1323118409
192.168.2.43.56380 > 172.217.10.129.443: Flags [P.], seq 1:518, ack 1
172.217.10.129.443 > 192.168.2.43.56380: Flags [.], ack 518
172.217.10.129.443 > 192.168.2.43.56380: Flags [.], seq 1419:2837, ack 518
```

With which output is it easier to see how much data has passed?

Absolute (-S option)

```
IP 192.168.2.43.56380 > 172.217.10.129.443: Flags [S], seq 1323118408
IP 172.217.10.129.443 > 192.168.2.43.56380: Flags [S.], seq 3183094287, ack
1323118409
IP 192.168.2.43.56380 > 172.217.10.129.443: Flags [P.], seq
1323118409:1323118926, ack 3183094288
IP 172.217.10.129.443 > 192.168.2.43.56380: Flags [.], ack 1323118926
IP 172.217.10.129.443 > 192.168.2.43.56380: Flags [.], seq 3183095706:3183097124,
ack 1323118926
```



Here is a different view of the same sequence of packets. The first set uses relative (the default) sequence/acknowledgement numbers, and the second set uses absolute sequence/acknowledgement numbers. The output has been truncated to show the pertinent details.

By default, tcpdump shows relative sequence numbers because 32-bit sequence number values may clutter the output and make it very difficult to discern how many bytes have passed at any specific point in the conversation. Part of the reasoning is that if you are looking at the sequence numbers in the middle of a session, you most likely want to figure out how much data has gone by. Relative numbers tell you this immediately and take up far less visual space. All sequence and acknowledgement numbers here that are related to the client have a solid underline. The ISN of the client is 1323118408, and the true ISN value of the server is 3183094287. The second display using absolute values shows the sequence numbers and acknowledgement numbers as they are found in the TCP header.

Look at the first session with relative values where tcpdump uses absolute sequence numbers on the SYN and SYN/ACK packets yet switches to relative numbers thereafter. For instance, the third packet sent by the client has an acknowledgement number of 1. This is relative to the server's sequence number. The absolute value is $3183094288 = 1 + 3183094287$ (server's ISN). Similarly, the beginning sequence number of the third packet set by the client is 1 and spans 517 bytes. This is relative to the client's ISN of 1323118408. Accordingly, in absolute value, this is sequence numbers 1323118409–1323118926 if added to the ISN for the client.

It takes a while to become accustomed to relative sequence and acknowledgement numbers. And if you prefer not to become all that familiar with them, there is always the -S command-line switch to display them as absolute numbers.

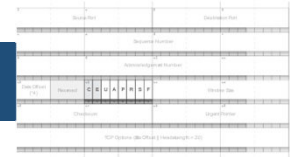


To follow along, enter the following commands for the absolute and then relative TCP sequence/acknowledgement numbers:

```
tcpdump -r tcp-fastopen.pcap -ntc 5
```

```
tcpdump -r tcp-fastopen.pcap -Sntc 5
```

TCP Flags



- **FIN:** Gracefully terminates a TCP session
- **SYN:** Establishes a new TCP session
- **RST:** Aborts a TCP session
- **PUSH:** Transmits data immediately:
 - Used when sender empties write buffer to signal to receiver to push the data to the application layer
- **ACK:** The existence of an acknowledgement number value is valid:
 - After initial SYN connection, should always be set
- **URG:** Existence of an urgent pointer value is valid
- **ECN-related:** Responds to an Explicit Congestion Notification (ECN)

TCP flags or *code bits* are bits in TCP offset 13 that are used to represent various signals in TCP communications. These flags are important because they inform the receiving host of the sending host's state in the TCP session. You may find many different, and sometimes erroneous, explanations of the purpose of some of the flags—specifically, the PUSH, ACK, and URG flags.

Let's start with some of the most straightforward explanations of flags. The FIN (finish) flag is used by a host to indicate that it has no further data to send. It is part of a graceful termination of a connection. Because the full-duplex TCP session is made up of two half-duplex connections, both sides of the session have to gracefully close by sending individual FINs in each direction, which are acknowledged by the other host.

The SYN (synchronize) flag is set when a host wants to establish a new session with a host. Because data must flow both ways in a TCP session, each host involved in the session must establish its own connection to the remote host. The SYN is set on the first two segments of a conventional three-way handshake, where both the client and server establish their own connection. The RST (reset) flag abruptly aborts a TCP session.

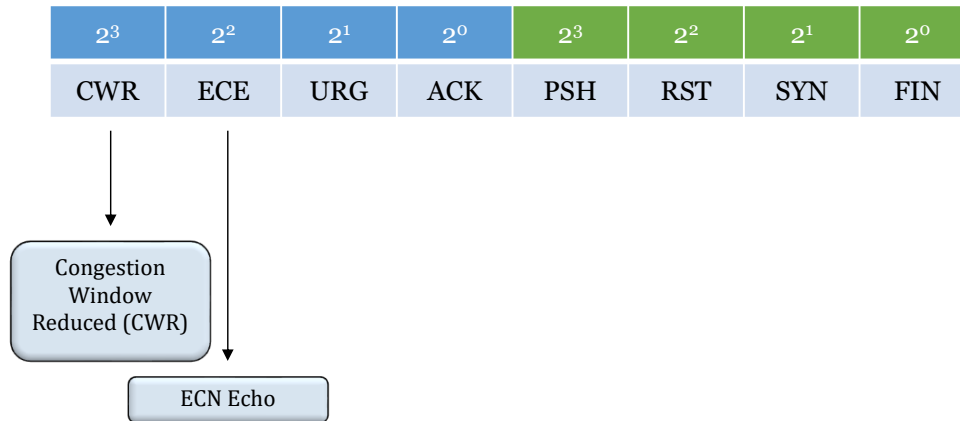
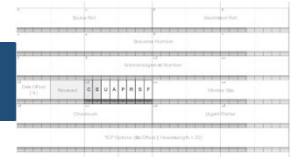
Each TCP endpoint has a receive buffer. It is possible that multiple segments can fit in the receive buffer, and this buffered data might not be delivered immediately to the receiving process. The PUSH bit instructs the receiver to deliver all pending data to the receiving process immediately.

The ACK flag simply indicates that the value found in the acknowledgement number field of the TCP header is valid. The ACK flag will *always* be set in normal traffic in every TCP segment after the initial SYN.

The URGENT flag is not often used, at least for legitimate purposes. Like the ACK flag, it signals that the urgent pointer is valid. The URG flag and associated urgent pointer value signal that there is some urgent or, according to Microsoft, *out-of-band* data that must be handled before other data. The urgent pointer value denotes the byte offset into the payload where the urgent data ends.

In some references, you will see the highest two bits marked as reserved. However, *Explicit Congestion Notification (ECN)* makes use of these previously reserved bits.

Explicit Congestion Notification (ECN)



ECN is a scheme that relies on the use of the 2 high-order bits in the TCP flags. ECN is a mechanism implemented within TCP that reduces IP congestion.

The high-order bit is known as the *Congestion Window Reduced* bit. It signals that the sender will reduce the speed at which data is sent. The bit to the right of that is the *ECN-echo* bit, and it is set to signal to a sender that it has caused, or has been involved in, congestion. If you would like to read more about ECN, reference RFC 3168. We will cover the basics here.

When we were discussing the IPv4 *type of service* byte, now known as the *differentiated services* byte (renamed as *traffic class* in IPv6), we introduced a future purpose for the 2 low-order bits in the TOS/differentiated service byte: Explicit Congestion Notification (ECN). The intent is for a router to notify a sender that there is congestion in the network and that it should reduce its sending rate.

How exactly does that occur? ECN, as defined in RFC 3168, is only implemented in a handful of protocols: TCP, SCTP (Stream Control Transmission Protocol), and DCCP (Datagram Congestion Control Protocol). We will only concern ourselves with the TCP implementation.

The 2 high-order bits of the TCP flag byte are used as fields for ECN. The ECN Echo bit will be turned on when TCP receives a packet that has the Congestion Experienced bits turned on in the differentiated services/traffic class byte in the IP header.

If TCP sets the ECN-echo bit, the purpose is to inform the other side of the conversation to reduce the rate it is sending data because there is congestion between the sender and receiver. Upon receipt of a TCP segment with the ECN-echo bit set, a host should reduce its *congestion window*, the sending buffer, by one-half. When it reacts in this manner, it turns on the Congestion Window Reduced (CWR) bit to inform the other side of the conversation that remedial action has occurred to reduce congestion. This bit is found in the high-order bit of the TCP flag byte.

ECN Flags

Differentiated Services Bits in IP								ECN Bits in TCP							
2 ³	2 ²	2 ¹	2 ⁰	2 ³	2 ²	2 ¹	2 ⁰	2 ³	2 ²	2 ¹	2 ⁰	2 ³	2 ²	2 ¹	2 ⁰
Codepoint / Precedence Bits				Server				CWR	ECE	URG	ACK	PSH	RST	SYN	FIN
SYN				0	0	0	0	1	1						
SYN/ACK				0	0	0	0	0	1						
ACK				0	1	0	0	0	0						
Normal, A->B				0	1	0	0	0	0						
Normal, B->A				1	0	0	0	0	0						
Congestion!				1	1	0	0	0	0						
Echo to sender				1	1	0	1	0	1						
Sender reduces window				1	1	1	1	1	1						

Three-way handshake for ECN-aware hosts

These will alternate in uncongested traffic.

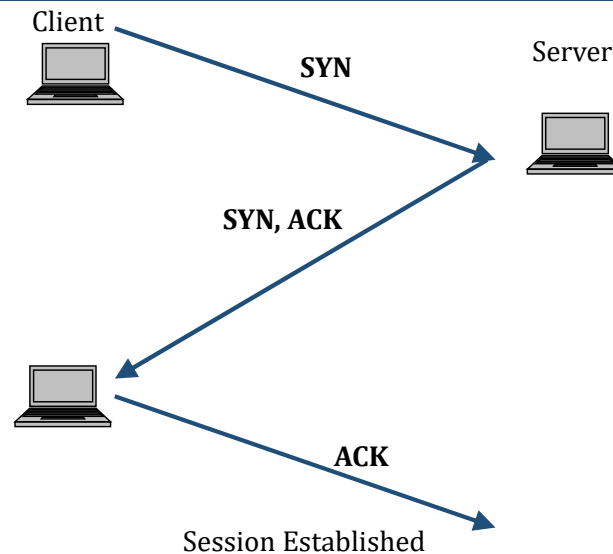
The IP header 2 low-order bits of the differentiated services bytes along with the TCP header 2 high-order bits of the TCP flag byte are coordinated to indicate the presence of ECN support and its state, if supported. Let's take a look at the possible settings. The first phase of ECN is both hosts signal that they are ECN-aware. Otherwise, if either or both are not, it makes no sense for a router to mark a packet congested if no possible remediation can occur.

When an ECN-aware host initiates a SYN, it sets a value of 00 in the 2 low-order ECN bits found in the differentiated services byte of the IP header. It sets a value of 11 in the 2 high-order ECN bits of the TCP flag byte. This informs the receiving host that the sender is ECN-aware. If the receiver is ECN-aware, it sets the SYN and ACK flags and sets the 2-high order bits to 01. When the original sending client receives these settings, it knows that the server is ECN-aware and clears the ECN TCP flag bits and sets a value of 01 in the differentiated services bit of the IP header.

The IPv4 header differentiated services bits are now used as the means to inform any router that the end hosts are ECN-aware. You may wonder why the ECN settings now switch to the IP header instead of remaining with the ECN bits of the TCP header to signal congestion. That is because the IP layer is what routers are able to examine. References indicated that a choice of values of either 01 or 10 is set in the IP differentiated services ECN bits to signal to routers that the communicating hosts are ECN-aware. You see the slide has a value of 01 in this field. This is because references state that the differentiated services bits are set to 01 on the ACK of the three-way handshake; yet no mention is made of a value of 10. So, we use 01 only to represent that no congestion exists for communicating ECN-capable hosts.

If congestion is detected and marked by the router, the ECN bits in the differentiated services byte are set to 11. And the receiving host must respond by setting the value in the TCP ECN flag bits to 01. The original sender receives this and informs the other host that it is responding to the congestion with a setting of TCP ECN flag bits value of 11.

The Three-Way Handshake



Step	Direction	Flag	Sequence	Window	Checksum	Options
1	Client to Server	SYN	1000000000	0	0	0
2	Server to Client	SYN, ACK	2000000000	0	0	0
3	Client to Server	ACK	2000000000	0	0	0

You see that establishing a TCP connection is almost ceremonial in nature, involving the *three-way handshake*. This is normally done before any data can be passed between the two hosts. What is depicted is the client initiating a connection to the server. The client designates a server port. Also, the client must designate and open a port for the server to send its communications.

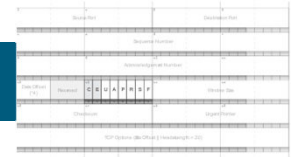
For the TCP three-way handshake, the client first sets a SYN flag to signal a request for a TCP connection to the server. Second, if the server is up and offers the desired service, and can accept the incoming connection, it sends the client a connection request of its own with the server's SYN flag set and, on the same packet, acknowledges the client connection request with an ACK. At this point, the second step of the TCP three-way handshake is completed.

Finally, if the client receives the server's SYN and ACK and still wants to continue the connection, it sends a final lone ACK to the server, acknowledging it has received the server's SYN. The connection has been established. Data can now be exchanged between the two. After the connection is established, every segment should have the ACK bit set. As much as possible, ACKs are *piggybacked* onto packets with data to minimize traffic instead of sending a packet with just an ACK.

This handshake requires that both hosts open up communication channels to the other host. TCP is full-duplex, meaning that either host can communicate with the other at any time, even at the same time. This is why each host has to synchronize (SYN) a session with the other during the three-way handshake.

As a tidbit of information, "SYN" is actually an abbreviation for *synchronize sequence numbers*.

Three-Way Handshake: Keys to the Connection



```
sans@sec503: tcpdump -r 3whs.pcap -c 3 -ntS
```

```
IP 192.168.11.62.36609 > 192.168.11.1.80: Flags [S], seq 1407137694, win 5840,
options [mss 1460,sackOK,TS val 169259530 ecr 0,nop,wscale 6], length 0
```

```
IP 192.168.11.1.80 > 192.168.11.62.36609: Flags [S.], seq 743334558, ack
1407137695, win 5792, options [mss 1460,sackOK,TS val 414457892 ecr
169259530,nop,wscale 1], length 0
```

```
IP 192.168.11.62.36609 > 192.168.11.1.80: Flags [.], ack 743334559, win 92,
options [nop,nop,TS val 169259531 ecr 414457892], length 0
```

2 ³	2 ²	2 ¹	2 ⁰	2 ³	2 ²	2 ¹	2 ⁰
C	E	U	.	P	R	S	F



This is the way tcpdump shows the packets involved in the session establishment, also known as the three-way handshake. When a TCP connection is established, there is a great deal happening in the three-way handshake. Not only is there signaling occurring using the TCP flags, but the keys to the connection are being established. These "keys" are the IP port pair, which includes the addresses of the two hosts and the ports that each host is using.

Notice the TCP flags involved in the three-way handshake. We'll discuss the various flag values in more detail in upcoming slides. TCP flags denote the state of the TCP connection and the exchanges of TCP segments. The SYN flag is used to synchronize, or establish, a session. The ACK, or acknowledgement, flag indicates the acceptance of data.

It's important to note how tcpdump indicates the status of the flags. Your eye might go to the "seq" and "ack" markers, but these aren't actually where the flags are. The flags are all displayed between the square brackets. Generally, these are very intuitive. [SF] would indicate that both the SYN and FIN flags are set. The ACK bit, however, is non-intuitive. Tcpdump, for reasons lost in the mists of history, represents the ACK bit with the single dot. So, [S.] indicates that SYN and ACK are set.

In a normal connection, we expect to see that the source and destination hosts remain the same, while trading positions depending on who is speaking. The same is true of the port numbers. Additionally, we expect to see that each host chooses its own, random, initial sequence number, which is acknowledged by the other end of the connection. It may seem strange at first, but the acknowledgement of the initial sequence number will *always* be the sequence number plus 1. This sounds unusual with the three-way handshake because no data is usually exchanged. However, a TCP segment with a SYN flag set consumes a byte, even though there is no attached payload or data. This is how it is defined in RFC 793 pertaining to TCP.



To follow along, enter the following command:

```
tcpdump -r 3whs.pcap -c 3 -ntS
```

The diagram illustrates the TCP three-way handshake process between a Client and a Server. It shows four laptops representing the devices involved. The process is as follows:

- Step 1:** The Client sends a **SYN** packet to the Server.
- Step 2:** The Server responds with a **SYN** packet to a middle laptop, which then forwards it to the Client.
- Step 3:** The middle laptop sends a **SYN/ACK** packet to the Server.
- Step 4:** The Server sends an **ACK** packet to the bottom laptop, which then forwards it to the Client.

A large blue circle encloses the middle three steps of the process, highlighting the exchange between the intermediate devices.

75

Pcap of Four-Way Handshake Session

```
192.168.1.104.52709 > 192.168.1.103.999: Flags [S], seq 2635457805
192.168.1.103.999 > 192.168.1.104.52709: Flags [S], seq 10, win 1234
192.168.1.104.52709 > 192.168.1.103.999: Flags [S.], seq 2635457805, ack 11
192.168.1.103.999 > 192.168.1.104.52709: Flags [.], ack 1
```

```
192.168.1.103.999 > 192.168.1.104.52709: Flags [P.], seq 1:17, ack 1
```

SEND THIS NOW!!!

```
192.168.1.104.52709 > 192.168.1.103.999: Flags [.], ack
```

```
192.168.1.103.999 > 192.168.1.104.52709: Flags [R.], seq 17, ack 1
```

Is this inbound or outbound?
Might this confuse
security tools?

SANS



4whs.pcap

SEC503: Intrusion Detection In-Depth 76

Let's look at tcpdump output from a four-way handshake session. The client 192.168.1.104 connects with a SYN to server 192.168.1.103, which listens on port 999 with the expected SYN flag set. The server responds to the SYN with a segment with a SYN of its own and an initial sequence number. Yet it does not set the conventional ACK flag, nor does it return an acknowledgement value.

Next, the client responds. It acknowledges the server's SYN but also sets its SYN flag again, awaiting the server's acknowledgement. The server acknowledges the client's SYN on the fourth segment. That means that the session is now established.

The server sends a test message of "SEND THIS NOW!!!" The crucial sign that the receiver accepts this data is in the next packet, where it acknowledges receiving the 17 bytes sent by the server. If a Snort rule looked for this content in an established session, it would not alert. The fix was to use a configuration on Snort's TCP stream preprocessor that included **require_3whs**. This enables it to analyze the direction of traffic properly. Snort 3 now even includes the ability to detect four-way handshakes, but this can still lead to confusion within your rules as to who the server is and who the client is. This also confuses stateful firewalls!

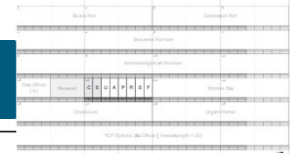
As it turns out, this behavior is defined by RFC! The writers of the RFC hypothesized about a potential situation that, as it turns out, never happens in TCP. Remember, this RFC was written in the 1970s, and they were trying to imagine everything that *could* happen. Their theory was that two hosts, independently, using the *same ports*, might decide to connect to each other at precisely the same time. This could create a situation where a host has sent a SYN, never received an ACK, but receives a SYN while waiting! This is defined in the RFC as the "Simultaneous Open." In practice, this never, ever happens in legitimate traffic. Would this, however, create any confusion for an IDS or a stateful firewall?

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

tcpdump -Ant -r 4whs.pcap

The output has been truncated and cleaned up to show relevant details.

Gracefully Terminating a TCP Connection



Server initiates a close with a FIN

192.168.11.1.80 > 192.168.11.62.36609: **Flags [F.]**, seq 743334927, ack 1407137695, win 2896, options [nop,nop,TS val 414459393 ecr 169259531], length 0

Client ACKs the server's FIN, initiates close with a FIN

192.168.11.62.36609 > 192.168.11.1.80: **Flags [.]**, seq 1407137695, **ack** 743334928, win 108, options [nop,nop,TS val 169263141 ecr 614459327], length 0

192.168.11.62.36609 > 192.168.11.1.80: **Flags [F.]**, seq 1407137695, ack 743334928, win 108, options [nop,nop,TS val 169263281 ecr 414459393], length 0

Server acknowledges client FIN

192.168.11.1.80 > 192.168.11.62.36609: **Flags [.]**, **ack** 1407137696, win 2896, options [nop,nop,TS val 414459393 ecr 169263281], length 0

A "graceful" termination is when one side of the connection, either the client or the server, initiates a close by setting the FIN flag. In theory, the exchange is supposed to go as follows:

1. Side 1 initiates a FIN to close its side.
2. Side 2 ACKs the FIN.
3. Side 2 closes its side with a FIN.
4. Side 1 ACKs the FIN.

In practice, this may or may not happen. In fact, in this output, steps 2 and 3 were combined into a single packet for efficiency. Why send two packets when one can send the same information? You may also witness busy hosts such as web servers combining the last segment that contains data, usually denoted when the PUSH flag is set with a FIN and ACK. This provides even more added efficiency.

Let's follow this sequence. The server, 192.168.11.1, initiates the FIN on a packet that has a sequence number value of 743334927. The FIN flag means that the initiator will send no more data, although it can accept data. The client, 192.168.11.62, acknowledges this with a value of 743334928. The client combines two steps in the flow of the graceful close into one by also setting the FIN flag to inform the server that it is now closing its side of the session. The sequence number on this segment is 1407137695. Finally, the server acknowledges the client's close, issuing sequence 1407137696 back to the client's FIN.

Neither the client nor the server sent any data, yet the subsequent acknowledgement numbers incremented the other side's sequence numbers by 1. Remember that the SYN and FIN segments were the exception to incrementing the sequence numbers by the number of payload bytes present. A segment with a FIN flag also consumes a sequence number.



If you'd like to follow along, enter the following command:

tcpdump -r 3whs.pcap -nt (last three records)

Aborted Session

Time	Source	Destination	Protocol	Length	Info
0.000000	10.1.3.7	10.1.3.35	TCP	60	8192 → 5840 [S] Seq=10 Win=0 Len=0
0.000000	10.1.3.35	10.1.3.7	TCP	60	5840 → 8192 [S.] Seq=3786702234 Ack=11 Win=5840 Len=0
0.000000	10.1.3.7	10.1.3.35	TCP	60	8192 → 5840 [.] Seq=11:20 Ack=3786702235 Win=8192 Len=9
0.000000	10.1.3.35	10.1.3.7	TCP	60	5840 → 8192 [.] Seq=20 Ack=3786702235 Win=8192 Len=18
0.000000	10.1.3.7	10.1.3.35	TCP	60	8192 → 5840 [P.] Seq=38 Ack=3786702235 Win=8192 Len=0
0.000000	10.1.3.35	10.1.3.7	TCP	60	5840 → 8192 [R.] Seq=38 Ack=3786702235 Win=8192 Len=0

```

10.1.3.7.45683 > 10.1.3.35.80: Flags [S], seq 10, win 8192, length 0
10.1.3.35.80 > 10.1.3.7.45683: Flags [S.], seq 3786702234, ack 11, win 5840,
options [mss 1460], length 0
10.1.3.7.45683 > 10.1.3.35.80: Flags [.], ack 3786702235, win 8192, length 0
10.1.3.7.45683 > 10.1.3.35.80: Flags [P.], seq 11:20, ack 3786702235, win 8192,
length 9
10.1.3.35.80 > 10.1.3.7.45683: Flags [.], ack 20, win 5840, length 0
10.1.3.7.45683 > 10.1.3.35.80: Flags [P.], seq 20:38, ack 3786702235, win 8192,
length 18
10.1.3.35.80 > 10.1.3.7.45683: Flags [R.], seq 38, ack 3786702235, win 8192,
length 0
10.1.3.7.45683 > 10.1.3.35.80: Flags [R.], seq 38, ack 3786702235, win 8192,
length 0

```

An abrupt close of an existing session can be accomplished using an RST flag. One thing you need to understand about an RST flag is that it is the last packet in the session. There should be no subsequent acknowledgements or segments sent. This needs a caveat: It assumes that the segment carrying the RST flag is correctly formed, meaning that it has an appropriate sequence number. You'll learn about TCP window sizes later, which may determine an appropriate TCP sequence number range. The RST must also have a good TCP checksum, which we'll soon discuss.

A relatively recent RFC 5961, "Improving the TCP's Robustness to Blind In-Window Attacks," has recommended a provision to mitigate reset attacks of existing TCP connections. It offers guidance to reset an existing TCP connection only if the TCP sequence number is the next expected one, as opposed to a range of TCP sequence number values related to the TCP window size that was previously acceptable. Additionally, it recommends that a host that receives a TCP sequence number on an RST that is within the acceptable window range return a "challenge ACK." This is a packet from the receiving host with the ACK flag set and the ACK value reflecting the next expected TCP sequence number.

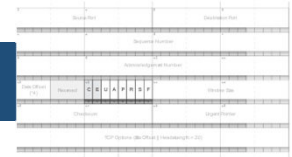
An RST flag is returned to the sender SYN whenever there is a closed port or when the segment is not part of an established session. For instance, say someone spoofed your IP address and sent a SYN to a remote host that listens on port 80. The remote host returns a SYN/ACK to your IP address. Your host issues an RST because it knows nothing about the session and has no indication that it sent an outbound SYN to the remote host.



To view the output, enter the command:

```
tcpdump -r rst.pcap -nt
```

Urgent Flag/Data



- URG flag indicates that a value in the urgent pointer field is valid
 - Urgent pointer value designates the byte offset from the current sequence number where the urgent data *ends*
- RFC 793 is ambiguous:
 - Is it the last byte of the urgent data? Or is it the byte after that? The RFC says both.
 - RFC 1122 seeks to clarify: It *definitely* points to the last byte of urgent data.
- Problem:
 - Some OSs point to byte in urgent data value.
 - Others point to byte after.
 - *Most end hosts drop this byte* while others ignore the urgent flag entirely!
- Interpretation dilemma for IDS/IPS; possible evasion

The TCP urgent flag and associated urgent value are rarely used legitimately today. The implementation uses the TCP urgent flag to signal that the value found in the urgent pointer offset field in the TCP header is valid. This offset indicates that there is data between the current sequence number in this packet and the offset indicated in the offset field that the receiving host should process before all other data that might be queued in its input buffer. The urgent data field in the TCP header is 16 bits and points to the byte offset where the urgent data ends.

The original TCP RFC, 793, introduces some significant ambiguity when it comes to interpreting this field. In section 3.1, it indicates that this offset points to the byte immediately *following* the urgent data (in other words, after it). Yet, on page 41 of that same RFC, it indirectly indicates that the urgent data ends *at* the value in the urgent pointer. Clearly, this introduces confusion and potential incompatibilities for network stack implementors! Fortunately, this feature is rarely used.

RFC 1122 seeks to clarify this, stating authoritatively that this pointer points at the last byte of the data, not to the byte following the data. Still, the damage has already been done.

Some implementations have interpreted the location of the urgent pointer to be the exact byte value specified in the urgent data field, whereas others have interpreted it as the byte following this value. So what's the fuss, and why do we care as analysts which interpretation a host adopts? Many receiving applications appear to drop the byte in the urgent data value. And this becomes a problem because if an IDS or IPS does not interpret the urgent data identically to how the end host does, we are presented with another means of evasion. Let's take a look at what happens with urgent data on the next slide, and we'll talk about the ramifications of IDS/IPS evasion.

Sample Urgent Data

IP 192.168.122.1 > 192.168.122.129.999: Flags[P.U],ack 1, win 8192, urg 1, length 26

payload:

ABCDEFGHIJKLMNOPQRSTUVWXYZ



Urgent data
pointer

netcat listener on 192.168.122.129

user@desktop:~\$ sudo nc -l -p 999

BCDEFGHIJKLMNOPQRSTUVWXYZ

netcat drops "A"

The problem is bigger than netcat; the client software must implement SIG_URG handler!

SANS



urg.pcap

SEC503: Intrusion Detection In-Depth 80

Here, host 192.168.122.1 sends host 192.168.122.129 a segment with 25 bytes, the letters of the alphabet. The urgent flag is set, and the urgent value is 1. This means that the payload byte at offset 1 (counting starts at 1) is considered to be the end of the urgent data.

A netcat listener on receiving host 192.168.122.129 displays the data it receives. As you can see, netcat has dropped the byte found at offset 1, the "A". Hypothetically, say we had an IDS rule that fires when the payload of "EVIL" is found in a stream. Say we craft two packets in the stream; the first has "EV". The second one has "AIL" with an urgent pointer set to 1. Most likely, the IDS/IPS sees "EVAIL", but the listening application sees "EVIL", unless it implements support for urgent data.

This whole issue gets more complicated because it is not the IP stack that determines how to deal with the urgent flag; it is the application that sends the data and the application that receives the data. In modern network stacks, the programmer must specify which interpretation of URG she chooses to use when sending data. The person writing the software on the other end of the connection must similarly decide how to handle these packets. Additionally, the person writing the software *must be aware that urgent data can arrive!* If the programmer is unaware, then she will not know that she *must* implement a SIG_URG handler and call *recv* with a special *msg_oob* qualifier to retrieve the urgent data. This is an issue for an IDS/IPS because it can't possibly know how every application will deal with urgent data.

As if that is not bad enough, you have interesting cases of whether a receiving application drops any data if the urgent value is 0. Also, according to the RFC, the urgent data pointer can be in a segment that is yet to be received. Some operating systems honor this; others do not. And what about the denial-of-service implications if the urgent pointer points to a large number that is outside the length of the current packet payload? Does the IDS/IPS set aside resources to track this or track an attack where all segments have large values in the urgent pointer? The precise handling of urgent data seems to vary not only by operating system, but also depending on the particular combination of the value of the urgent pointer and whether the urgent data is all contained within the current packet or not. We have even seen at least one operating system (MacOS) that seems to completely disregard the urgent bit and pointer, simply inserting the urgent data directly into the data stream.

To view the output, enter the following command (the packet of interest is the fourth one):

tcpdump -r urg.pcap -nt

TCP Flags and Payload Allowed

Source Port	Destination Port
Sequence Number	Destination Sequence Number
Window Size	Window Scale
Checksum	Urgent Pointer
Options	Padding
Payload	

- RFC 793 allows payload with the following flag combinations:
 - **SYN flag set:** Data on SYN, seen on TCP Fast Open sessions
 - **PUSH/ACK flags set:** Most common combination of flags
 - **ACK flag alone set:** Not abnormal with ACK alone
 - **URG with PUSH/ACK:** Rarely seen*
 - **FIN with PUSH/ACK:** May be observed, sent from busy web servers

An IDS/IPS must begin to analyze payload only after a TCP session has been established. In the early stages of IDS development, attacks such as *Stick* and *Snot* attempted to cause problems for an IDS that did not ensure that a TCP session had been established before analyzing payload. The result was that the IDS fired false positives to the point of distraction for the analyst. Nowadays, all IDS/IPS solutions keep track of TCP session state as they analyze payload data.

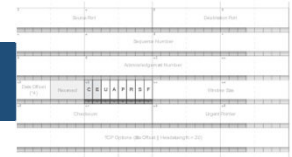
RFC 793 offers a provision for a host to send data on the SYN packet. Until the introduction of TCP Fast Open, data was rarely sent or accepted when sent with the SYN. After the TCP session has been established, the ACK flag should be set on every segment. The PUSH and ACK flags on a segment with payload is the most common combination.

It is not necessary to set the PUSH flag to send data because the ACK flag alone is sufficient and not unusual to see. As we discussed earlier, the URG flag is rare these days but is acceptable when sending payload. Please note that the reason it is rarely seen with a PUSH/ACK is that *the urgent bit is exceedingly rare to see at all!* In fact, if you were to send an urgent packet, it makes complete sense that it would be a PUSH/ACK/URG!

Finally, the FIN and ACK flags with or without the PUSH flag set are valid combinations for sending payload. You may see this behavior from web servers that attempt to minimize the separate segments by combining the session close and the last data transmission.

Just because RFC 793 supplied the guidelines for proper TCP flag settings does not mean that all TCP/IP stack implementations adhere to this guidance. A good IDS/IPS follows the RFC guidelines but also attempts to understand the aberrations when it does payload analysis.

TCP Connection Retransmissions



- Repeated attempt to connect to TCP port
- If no RST received, connection attempted again:
 - Destination host down; no ICMP message sent
 - Packet-filtering device silently dropping
 - RST sent, but host doesn't receive it
- Number of additional retries sent; operating system dependent
- Finally gives up connection attempt

What if a TCP connection is attempted, yet the sending host attempting the connection doesn't receive a response from the receiving host? The intended receiver host might not respond for a number of reasons.

This may occur during an attempted session establishment because the receiver may not be up or may not exist. A router may attempt to deliver an ICMP message that the receiver is unreachable. But if the router has been silenced from delivering unreachable messages, the sending host will never know that there is a problem. Also, the receiver might be sitting behind some kind of packet-filtering device that blocks the connection inbound, yet silently drops the connection without informing the sending host.

It is also possible that the receiving host responds positively (SYN/ACK) or negatively (RST/ACK), yet for some reason the sending host doesn't receive these replies.

Additional attempts or retransmissions will be made to contact the receiving host in situations like this. The number of retransmissions and the time intervals between attempts vary by operating system. Eventually, the sending host ceases sending the connection attempts.

TCP Retries

Time	Source IP	Destination IP	Source Port	Destination Port	Protocol	Length	Flags	Sequence	Window	Options	Checksum	ICMP Type	ICMP Code	ICMP Data
10:18:31.693163	192.168.11.62	192.168.11.46	33187	80	TCP	60	S	3814660388	5840	mss 1460,sackOK,TS val 235028765 ecr 0,nop,wscale 6	0			
10:18:34.689451	192.168.11.62	192.168.11.46	33187	80	TCP	60	S	3814660388	5840	mss 1460,sackOK,TS val 235029515 ecr 0,nop,wscale 6	0			
10:18:40.689451	192.168.11.62	192.168.11.46	33187	80	TCP	60	S	3814660388	5840	mss 1460,sackOK,TS val 235031015 ecr 0,nop,wscale 6	0			

Which set are retries?

```
10:18:31.693163 IP 192.168.11.62.33187 > 192.168.11.46.80: Flags [S], seq 3814660388,
win 5840, options [mss 1460,sackOK,TS val 235028765 ecr 0,nop,wscale 6], length 0
10:18:34.689451 IP 192.168.11.62.33187 > 192.168.11.46.80: Flags [S], seq 3814660388,
win 5840, options [mss 1460,sackOK,TS val 235029515 ecr 0,nop,wscale 6], length 0
10:18:40.689451 IP 192.168.11.62.33187 > 192.168.11.46.80: Flags [S], seq 3814660388,
win 5840, options [mss 1460,sackOK,TS val 235031015 ecr 0,nop,wscale 6], length 0

11:01:13.714780 IP 192.168.11.62.1024 > 192.168.11.46.80: Flags [S], seq 933503970, win
8192, length 0
11:01:15.856423 IP 192.168.11.62.1025 > 192.168.11.46.80: Flags [S], seq 933740120, win
8192, length 0
11:01:17.883880 IP 192.168.11.62.1026 > 192.168.11.46.80: Flags [S], seq 933573291, win
8192, length 0
```



Suppose you were looking at traffic output and you saw the above sets of connections. The first represents several connections from 192.168.11.62 to 192.168.11.46 to port 80. The second represents another set of connections from 192.168.11.62 to 192.168.11.46 to port 80. Do you have any idea which set contains multiple different connection attempts and which set represents retries or retransmissions?



The first set represents the retry and the second a SYN scan. Both have the same source and destination IP addresses and destination port 80. The retry is different in several ways. Many operating systems use something called a *back-off timer*, which dictates the time interval between each retry. The operating system above doubles the retry time interval for each successive try. There is a 3-second difference between the first and second retry and a 6-second difference between the second and third. Other operating systems may use a constant time interval.

Various operating systems generate a different number of retry attempts, not just an initial one followed by two retries, as shown here. The packet essentially remains the same for each retry; the source port and sequence numbers do not change.

Compare that with a scan. It's hard to tell with three records only, but there is no predictable back-off timer. The source ports change, as do the TCP sequence numbers.

To view the output, enter the following command:

```
tcpdump -r retry-or-synscan.pcap -n
```

TCP Delivery Failures

- How does the receiver let the sender know data has not been received?
 - Lost or delayed TCP segment from the middle of a stream:
 - Duplicate acknowledgement number from receiver
- No acknowledgement by receiver:
 - Retransmission timeout by sender

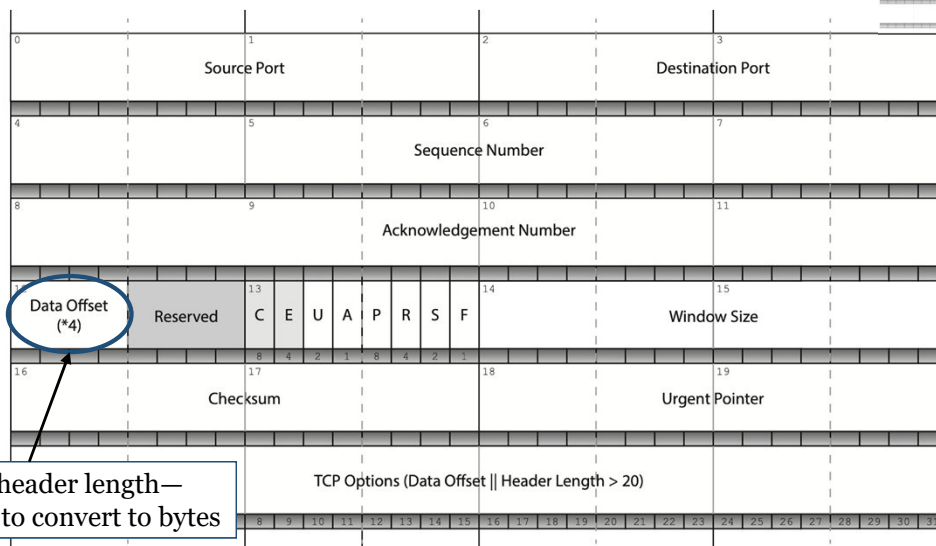
What happens with the TCP flags and the sequence numbers if data does not arrive? Can TCP detect gaps in data?

If the receiver gets a TCP segment with a sequence number greater than the one it expects, it will not acknowledge the out-of-order sequence number. Instead, it sends back an acknowledgement of the next expected sequence number (in other words, it tells the sender which byte of data in the sequence of bytes it expected to receive). Even when the TCP option known as the selective acknowledgement is used, the receiver TCP can acknowledge only the next expected chronological TCP sequence number. The selective acknowledgement informs the sender TCP that it has received out-of-order sequence numbers; however, it still returns the next expected sequence number in the acknowledgement field. We will look at the behavior of SACK in our TCP options discussion later in this section.

So, if a host receives a sequence number higher than it is expecting, it will send an ACK for the sequence it was expecting. It is possible for there to be no acknowledgement at all by the receiver. For instance, if the last TCP segment in a stream is lost, there will be no duplicate acknowledgement numbers to inform the sending host. Instead, the sending host has its own mechanism to deal with this, known as a *retransmission timeout*. If a sending host does not receive an acknowledgement before a timer expires, it retransmits the unacknowledged segment.

How long is the retransmission timer? This is dependent on each individual connection. TCP is an adaptive protocol; it is aware of the network conditions. It tries to time the round-trip packet exchange to know what is considered an acceptable or normal transmission time. This value should be different for hosts on a local network versus hosts on distant networks. Therefore, this value is not constant for all sessions or even throughout the same session.

TCP Header Length Field



There is a single length field in the TCP header. This represents the length of the TCP header. Like the IPv4 header, the TCP header can have options. The TCP header is 20 bytes without TCP options. Unlike IPv4, TCP options are common and more often than not used in TCP sessions. Actually, it is standard for any TCP segment with the SYN flag set to have the TCP option maximum segment size field and value present.

This TCP header length field is found in the high-order nibble of the 12th byte offset in the TCP header. One final similarity between the IPv4 and TCP header lengths is that they must be multiplied by 4 to convert to bytes. It is necessary to know how long the TCP header is so that you know where the TCP header stops and where the TCP data starts. You may see the header length called the "offset" in some TCP header layouts.

TCP Header Length

10.10.10.10.63220 > 10.10.10.11.139: Flags [S] seq 776342897, length 0

4500 0028 e34f 0000 3a06 e534 0a0a 0a0a

0a0a 0a0b f6f4 008b 2e46 0d71 0000 0000

0 1 2 3 4 5 6 7 8 9 10 11

5002 0c00 b85f 0000

12 13 14 15 16 17 18 19

TCP header length=5*4=20

Found in high-order nibble of
12th byte offset in TCP header

Look at the TCP header length field. You can see it is the first hex character or high-order nibble of the 12th byte in the TCP header. In this case, you find a value of 5 that we multiply by 4 for 20 bytes, which represents a standard TCP header with no options. As with the IP header, computing this value is crucial if you want to examine the TCP payload, requiring you to determine where the TCP header stops and the TCP data begins.

TCP Header Length with TCP Options

```
10.10.10.10.3088 > 10.10.10.11:139 flags[S], seq 1212214992, win 32120,
options[mss 1460,sackOK,timestamp 7748460 0,nop,wscale 0]
```

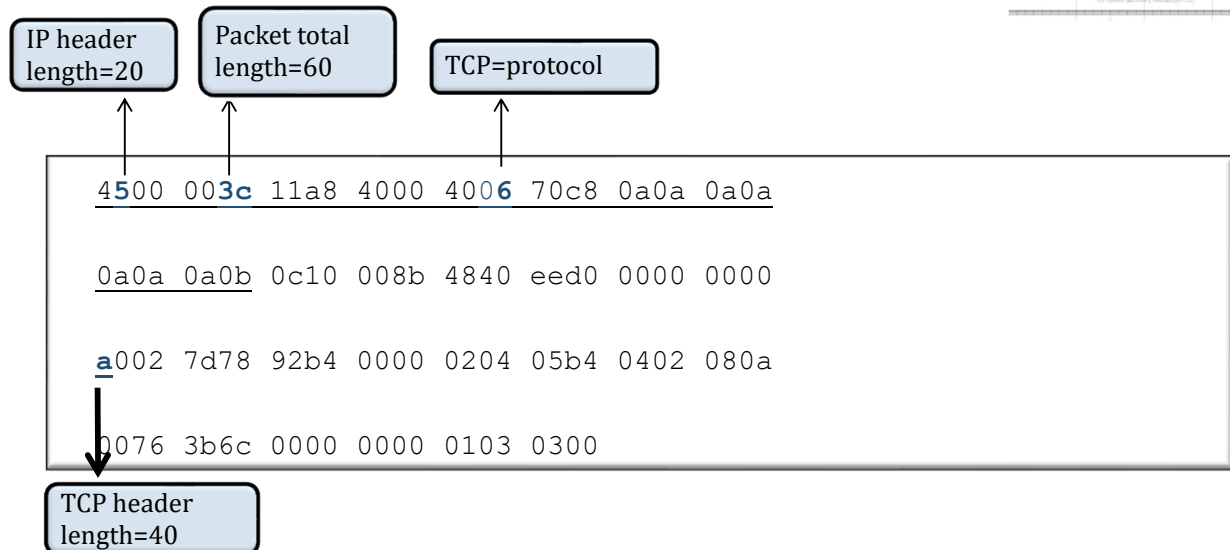
```
4500 003c 11a8 4000 4006 70c8 0a0a 0a0a
0a0a 0a0b 0c10 008b 4840 eed0 0000 0000
a002 7d78 92b4 0000 0204 05b4 0402 080a
0076 3b6c 0000 0000 0103 0300
```

TCP header length=10*4=40

Now look at the above packet. You see that it has a TCP header length of 0xa, which is a decimal 10. This value multiplied by 4 indicates a TCP header length of 40 bytes. The underlined portions of this output represent the TCP options values. These include options maximum segment size of 1460, sackOK, timestamp, and a wscale. We will discuss these TCP options later in this section.

Like the IPv4 options, the entire set of TCP options must end on a 4-byte boundary, which is why you see a *nop* in the middle to add a single pad byte. Perhaps you are wondering why that *nop* value is not the last one because it stands to reason that might be the best place to pad. Most current operating systems place the pad byte(s) before TCP option(s), although there is nothing in standards that preclude the placement of the pad byte(s) after a TCP option(s). Actually, if you run *nmap* to perform a remote operating system scan, it uses the order and placement of the pad bytes as one of the criteria for determining the operating system. As you expect, the TCP options are stored in the TCP header after the 20 bytes of standard TCP header values.

Packet Dissection Revisited

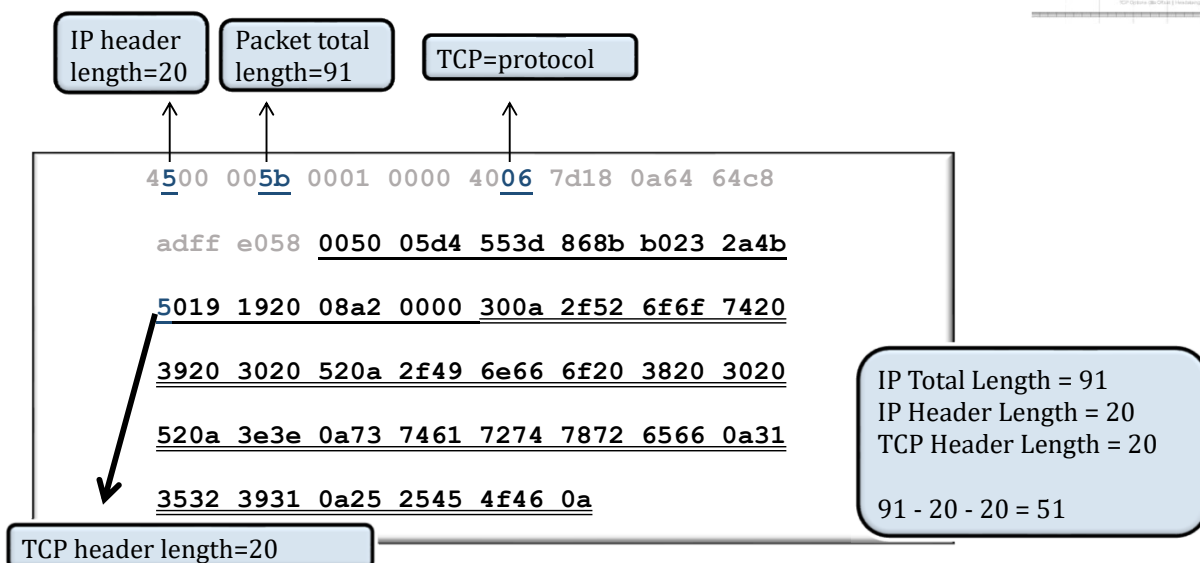


Suppose someone handed you this hexadecimal output and told you that this was an IPv4 packet. How would you go about making any sense of it? Let's use some of the packet dissection tricks you learned about the IPv4 header and add the new knowledge about the TCP header to interpret this hexadecimal.

First, we have a 20-byte IP header length because there is a 0x5 in the lower nibble of the zero-byte offset of the IPv4 header. Next, the entire packet length is 60 bytes because we have a value of 0x003c in the second and third bytes offset from the IPv4 header. And the transport layer is TCP because the value of 6 is in the protocol field in the 9th byte.

We have figured out that the IPv4 header ends after 20 bytes, denoted by the underlined portion of the above output. The TCP header begins immediately after that. If you superimposed a template of a TCP header over the values in the above TCP header, you could interpret all the fields, such as the source and destination ports, the flag bytes, and so on. We know from the previous slide that we have a TCP header with 40 bytes. For now, we've dissected a packet with no data following the TCP header. This is a packet with a SYN flag set as shown in the tcpdump output as "flags [S]" in the previous slide. This is why there is no data.

TCP Header and Data: How Many Bytes of Data?



Now that you've had a chance to dissect some packets on your own, you understand the need for the length values. They indicate where the given protocols, header, and values begin and end. This is essential for you to dissect the packets just as it is for the TCP/IP stack to de-encapsulate those same packets.

Remember, in Section 1, we discussed the different types of lengths that might be seen in different protocols? We have some variable-length fields such as the lengths for the IPv4 header, the IPv4 total packet, and the TCP header. Yet, there is no dedicated field for the TCP data length. You might argue that this isn't needed because all the bytes after the TCP header are data bytes. But, for us to validate this, we can derive the length. The formula for the number of TCP data bytes follows:

$$\text{IP total packet length} - \text{IP header length} - \text{TCP header length} = \text{TCP data bytes}$$

Therefore, using this packet, you have:

$$91 - 20 - 20 = 51$$

Your Turn

How many bytes of TCP data are in the following packet?

```
4510 005a 3695 4000 4006 6c69 c0a8 0b3e
c0a8 0b01 8c6e 0050 f30e 8c5f 9814 ce41
8018 005c 5b2d 0000 0101 080a 0a0d c084
18b0 8a10 4141 4141 4141 4141 4141 4141
4141 4141 4141 4141 4141 4141 4141 4141
4141 4141 4141 4141 0d0a
```



See if you can figure out how many bytes of TCP data are in the packet displayed here.



When you finish, you can verify your answer by entering the following command:

```
tcpdump -r push-short.pcap -nt
```

The length value appears in tcpdump, reflecting the number of bytes in the TCP data.

Answer

IP 192.168.11.62.35950 > 192.168.11.1.80: Flags [P.], seq 4077816927:4077816965, ack 2551500353, win 92, options [nop,nop,TS val 168673412 ecr 414222864], length 38

IP header
length=20

IP packet
length=90

```

4510 005a 3695 4000 4006 6c69 c0a8 0b3e
c0a8 0b01 8c6e 0050 f30e 8c5f 9814 ce41
8018 005c 5b2d 0000 0101 080a 0a0d c084
18b0 8a10 4141 4141 4141 4141 4141 4141
4141 4141 4141 4141 4141 4141 4141 4141
4141 4141 4141 4141 0d0a

```

TCP header length=32

IP Total Length = 90
IP Header Length = 20
TCP Header Length = 32

$90 - 20 - 32 = 38$

SANS



push-short.pcap

SEC503: Intrusion Detection In-Depth 92

Tcpdump indicates that there are 38 bytes of data. There is a standard 20-byte IPv4 header, a total packet length of 90, and a TCP header with options 32 bytes in length.

$90 - 20 - 32$

Therefore, our formula indicates the presence of 38 TCP data bytes. If you count the underlined data bytes, there are 38 bytes present.

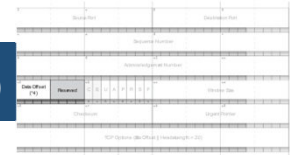


You can verify your answer by entering the following command:

tcpdump -r push-short.pcap -nt

By the way, congratulations on your packet dissection skills! Many of you entered this course without the knowledge and practice to do so. Look how far you've advanced already.

Weird Lengths: TCP Segmentation Offload (TSO)



- Normally, segmentation of TCP data is done by the kernel
- TSO moves the segmentation process to the NIC driver to lessen load from CPU and kernel
- Tcpdump and Wireshark examine packets from the kernel, large length packets seen
- **MAKE SURE THIS IS TURNED OFF ON PACKET COLLECTION SENSORS!!**

```
192.168.1.51.32924 > 192.168.1.8.5001: Flags [.], seq 2896:8688, ack 1, length 5792
192.168.1.51.32924 > 192.168.1.8.5001: Flags [.], seq 8688:11584, ack 1, length 2896
192.168.1.51.32924 > 192.168.1.8.5001: Flags [.], seq 14480:23168, ack 1, length 8688
192.168.1.51.32924 > 192.168.1.8.5001: Flags [.], seq 31856:43440, ack 1, length 11584
192.168.1.51.32924 > 192.168.1.8.5001: Flags [.], seq 55024:72400, ack 1, length 17376
192.168.1.51.32924 > 192.168.1.8.5001: Flags [.], seq 72400:86880, ack 1, length 14480
192.168.1.51.32924 > 192.168.1.8.5001: Flags [.], seq 86880:107152, ack 1, length 20272
192.168.1.51.32924 > 192.168.1.8.5001: Flags [.], seq 150592:173760, ack 1, length 23168
```



A feature known as *TCP Segmentation Offload (TSO)* moves the process of segmentation of TCP data into individual packets from the kernel to the NIC driver. NICs can have *Application Specific Integrated Circuits (ASICs)* that are customized to perform specific tasks, making them far more efficient because they lighten the load on the CPU and kernel.

One example of shifting a process to the NIC is performing segmentation of TCP stream payload into separate packets. The issue for analysts is that tools such as tcpdump and Wireshark process the packets when they are still in the kernel and before TCP segmentation offload occurs. This means that you may see unusual packet lengths that are too large for any MTU.

Be aware that if you see these exceptionally large lengths for TCP segments, check to see if the host that is capturing the packets has TSO turned on. A tool named ethtool can do this for you with the following command:

ethtool -k eth0 (where eth0 is the listening interface)

Segmentation offload can be disabled, although it is not recommended for production servers. On the other hand, you should *absolutely* turn this off for any and all IDS packet collection engines. If your NIC is reassembling packets for you, you have no idea what was in the *original* frames that were on the wire!

Another example of shifting processing to the NIC is checksum computation of packet protocols (IP, TCP, etc.).

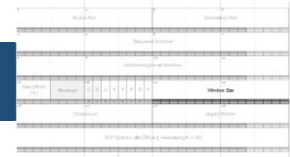


To see the slide output, enter:

tcpdump -r tsoffload.pcap -nt

(Edited to show pertinent details of longer length packets only)

TCP Window Size



Represents receiving host's TCP buffer size for connection

- The sender may send only the window worth of bytes *without* receiving an ACK.

Acts as flow control:

- Window size dynamically changes as data is received:
 - May go down, indicating data is not being pulled out of the receive buffer immediately
 - If it goes up or stays the same, it moves the "window," allowing more data to be sent.
- Window size of 0 tells sending host to temporarily cease sending data
 - When host can receive additional data, window size increases to greater than 0
- Can be made larger through the wscale scaling option

Initial window size can be used for OS fingerprinting

The TCP window size is the way that a receiving host informs the sending host of the current buffer size for data received for that connection. This is a flow control mechanism because it is dynamic.

The receiving window buffers data that is received and pushes it up to the associated application for the port in use. The window size becomes smaller when the application is not yet ready for more. If the receiving buffer ever becomes full, the window size becomes 0, informing the sending host not to send any more data. After the receiving host has processed some of the data and freed up room in the buffer for more, it sends a window size update with a value for the amount of data it can currently accept. This informs the sending host to resume transmission of data.

The control of flow for TCP sessions is mostly done by the receiving host through the window size. We have a tendency to assume that the sender is the one controlling the flow of data across the network, but for the most part, the receiver is in full control, using the window size to control the flow of data.

You have likely noticed that the window size is a 16-bit field, giving us a maximum value of 65535. This value, 64k, isn't very much by today's standards. We can easily fill that window very quickly! Can we specify a larger buffer? Yes. This is what the wscale option allows us to do. This is frequently described as a "multiplier," but it is really a "scaler." What's the difference? We don't multiply the window size by the value of the wscale option; instead, the value in wscale is used to *left shift* the value in the window size. This allows us to create very large buffers.

Initial window sizes are used by nmap to determine the operating system. Many operating systems select different initial window sizes, and this is used to help fingerprint the operating system.

Window Size and Throughput

Have you ever wondered why a very high-speed link seems to max out at much less than maximum speed?

- Throughput = Window Size (in bits)/Latency (in seconds)
- 64k Window, 2 ms latency: $524280/0.002 = 262,140,000$ (262.1 Mbps)
 - Even on a 40 Gbps link, you will *never* see more than 262 Mbps!

How do I know what window size to use?

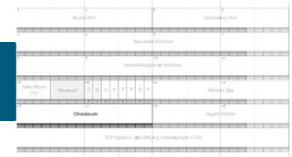
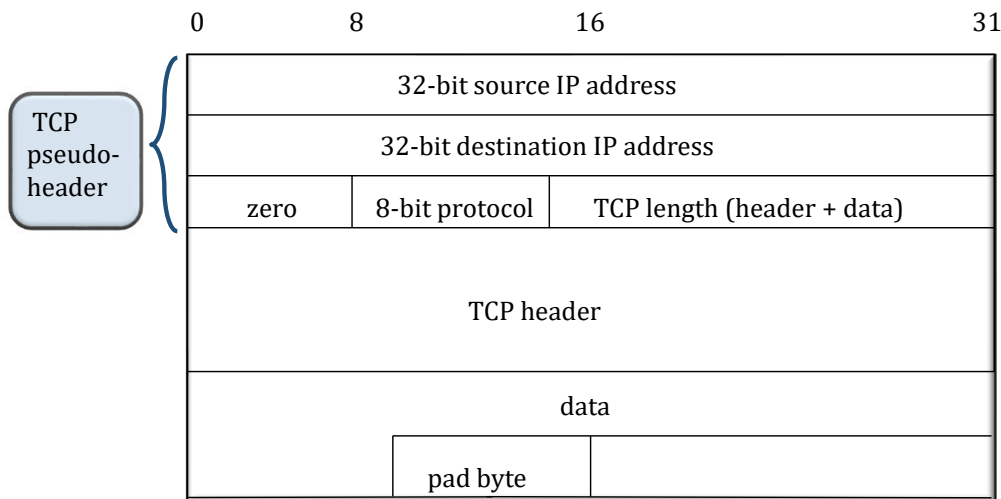
- Link speed * RTT = Ideal Window Size
- 1 Gigabit link, 4ms RTT = $1\text{Gbps} * 0.004 = 4,000,000$ bits = 500,000 bytes
- 1 Gigabit link, 20ms RTT = $20,000,000$ bits = 2.5 megabytes
 - Window scale option allows you to exceed 64k

Now that we know what the window size field does, we need to talk about some of the implications of the window size. One of the most direct ways that you can observe the impact of the window size field is to try to perform a large file transfer across a network. Have you ever noticed that your initial throughput looks quite good, but after just a few seconds, the overall throughput drops dramatically, often to less than half of the speed of the link? Have you ever wondered why this happens? While it is true that the transfer will slow or even stall if the receiver reports that its window is full, even if the receiver *never reports a window size of zero*, the connection will slow or even stall! The window size is responsible for this.

Since the receiver's window size tells the sender how many bytes may be sent without an ACK, the sender will initially saturate the link with however many packets will fill the currently advertised window. If the window size advertised is 64k, the sender will send the first 64k of data... and then stop until it receives an ACK for some portion of that data that shows a window size greater than zero. Can you see how, even on very fast networks with very low round-trip times (RTT), the sender may be waiting around for an ACK to get back?

You can work out precisely what the maximum throughput for a connection will be based on the RTT, the link speed (of the slowest link between the hosts), and the window size. Conversely, you can work out the ideal window size that you should configure based on the RTT and the link speed! If you follow this logic through, however, you will quickly realize that modern networks will saturate a 64k window size, the maximum, almost instantly. This is where TCP options come in. We will discuss the window scale option briefly in a few pages.

TCP Checksum



As mentioned previously, the embedded protocols have checksums as well. These cover the embedded protocol header and data. Unlike the IPv4 IP checksum, these are end-to-end checksums calculated by the source and validated by the destination host only. The TCP checksum has been chosen to show what the embedded protocol checksums look like. The same checksum theory applies to UDP and ICMPv6.

The embedded protocol checksums for TCP and UDP are computed using a pseudo-header in addition to the embedded protocol header and data. A pseudo-header consists of 12 bytes of data depicted in the slide, the source and destination IPs, the 8-bit protocol found in the IP header, and the embedded protocol header plus data length.

Why is the pseudo-header necessary? This is an additional check that is used by the receiving host to validate that the IP layer has not accidentally accepted a packet destined for another host. If there is some corruption that occurs in transit, the validation of the IPv4 IP checksum may or may not discover this, but some fields from the IP header are included in the pseudo-header checksum computation to protect against this.

Just a heads-up: Be aware that if you do some kind of packet manipulation such as with Scapy, checksums may come back to haunt you, especially the concept of the pseudo-header information. For instance, say you change an IPv4 address in a TCP packet. It is more obvious that the IP header checksum must be recomputed because a field in the IP header has changed. But the TCP checksum will be incorrect unless you recompute it as well.

```

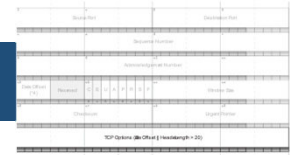
graph TD
    A[Valid IP checksum?] -- No --> B[Discard packet]
    A -- Yes --> C[Valid TCP checksum?]
    C -- No --> B
    C -- Yes --> D[Send to application layer]
    D --> E[Computer Icon]
  
```

The TCP layer performs this same validation process on the checksum value found in the TCP header. If the checksum is valid, the packet is passed to the application. If the checksum is invalid, the packet is discarded. There is no reset returned by the receiving host or any other error because the received packet was damaged.

Assume that host 10.2.3.5 exists and is listening on the destination port in the TCP header. When the corrupted packet arrives, the IP layer validates the IP checksum. It is correct, so the packet is pushed up to the transport layer (TCP). Since the TCP checksum in the header was calculated using the original destination address from the original IP header, the checksum will be invalid! The packet will be discarded silently by the TCP layer.

© 2021 David Hoelzer
<https://t.me/learningnets>

TCP Options



Included at the end of TCP header:

- TCP Fast Open
- Maximum Segment Size (MSS): Sort of an MTU for TCP
- Window scale: Specifies how far left to shift the window size, maximum value is 14
- Timestamp: Carries a "timestamp" for each segment
- Selective acknowledgement: Allows noncontiguous segments to be acknowledged
- Many more!
- No operation: Pad options to 4-byte boundaries
- End of list option: Pad final option to 4-byte boundaries

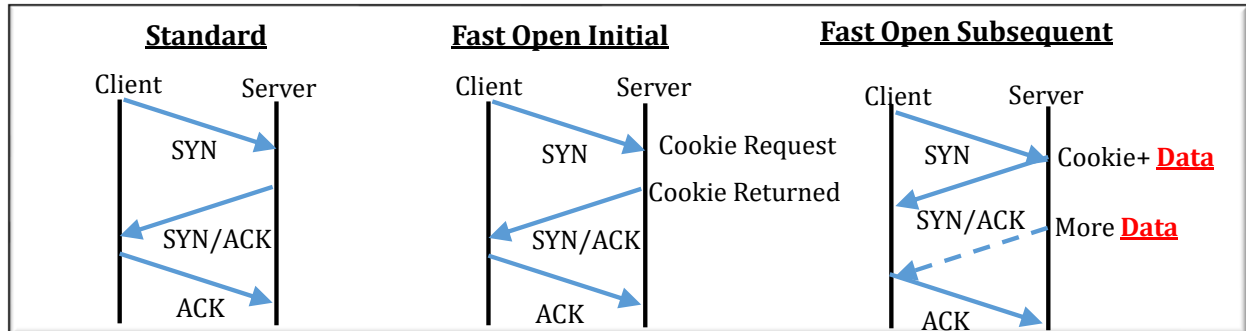
Can be used to do OS fingerprinting and for evasion attacks

TCP options are optional parameters appended to the TCP header. IANA maintains a list of the currently well-known options at <https://www.iana.org/assignments/tcp-parameters/tcp-parameters.xhtml>. We will discuss only a small selection of the more common options that you will see regularly.

- **TCP Fast Open** is a relatively new adjustment that seeks to optimize connections by reducing the number of packets required in a connection. We will talk about this in the next few pages.
- **MSS**, or *Maximum Segment Size*, may only be used on the SYN and represents the largest TCP payload that a host wants to send. Remember that the total IPv4 packet is at least 40 bytes more than this, accounting for the IP header and the TCP header. The optimal value for this is $MTU - 20(IP) + 20(TCP) + X(TCP\ Options)$. Since these values are determined on the SYN, the host must commit to a maximum, estimating required options. Both the client and server will send their choice for an MSS. Both hosts will use the lower of the two values for the ensuing conversation. The minimum supported MSS in IPv4 is 536 bytes, while it is 1220 in IPv6.
- The **wscale** option was discussed previously when covering the Window Size header field.
- The **timestamp** is a mechanism to time and compute round-trip time for TCP exchanges when both hosts support the option. This is used to compute the retransmission timer that helps recover from packet loss. The timestamp is also used to make sure an old segment does not accidentally get included in a current exchange. The TCP timestamp in a segment must be greater than or equal to the previous in-order timestamp sent.
- **SACK** is another relatively new option that seeks to optimize TCP communications and reduce retransmissions for out of order TCP segments. We will look at this option in a few pages.
- The **EOL** and **NOP** options are used to either terminate a list of options or otherwise pad options out so that they always fall on 4-byte boundaries.

TCP Fast Open

- Many changes are being driven by large content delivery networks:
 - HTTP traffic characterized by multiple simultaneous sessions for different elements
 - Needless packets consume "time slots," delaying other connections
- Solution: TCP Fast Open



As you know, TCP session establishment via the three-way handshake consists of three different packets. When simultaneous sessions are established on a busy server, in aggregate and in volume, performance issues can result. This is especially so for HTTP servers that often consist of small concurrent sessions, each returning a single object, such as an image or code. While the *number* of connections is being addressed through other changes to how network protocols work, the *number of packets* creates a different problem: For every packet traversing the network from a source host to this specific web server, that packet is taking up space, time, or a "time slot" that another packet cannot occupy. If we could eliminate needless packets, then, it would allow for more efficient use of the network.

The solution to the inefficient time and resource consumption required for establishing multiple related sessions is known as *TCP Fast Open*. Both the client and server must support it to be used.

The initial session between the client and server supporting Fast Open uses the same three-way handshake, but the client SYN carries a *cookie request* as a TCP option, and the server SYN/ACK returns a cookie in a TCP option that is cached by the client. Subsequent sessions that begin shortly thereafter send the cached cookie on the SYN as a TCP option. It turns out that TCP has always allowed a host to send data on the SYN, but according to the original RFC, this data would not be processed until the connection completes. TCP Fast Open modifies this behavior. If a connection is established using a known cookie and the SYN contains data, the receiver can begin to process that data *immediately*, without waiting for the connection to complete. This could potentially result in a TCP conversation that is only six packets long, yet a complete web request was serviced: SYN with a GET, SYN/ACK with the data requested, FIN/ACK, ACK, FIN/ACK, and ACK. (Theoretically, you could simplify this further, but in practice, the four-way close still occurs today.)

The cookie is created by the server. The mechanics of these are not particularly important since this really isn't an authentication token. While the RFC suggests that this *might* be cryptographically generated, there is no requirement that such an algorithm is used. The purpose of the cookie is to allow a server to have some reasonable assurance that the client will complete the three-way handshake. The server is responsible for authenticating the cookie for the client on subsequent sessions. The efficiencies are realized with subsequent sessions when the data is sent by both the client and server before the three-way handshake is complete, pending acknowledgement by both sides to accept the data to push to the TCP layer.

TCP Fast Open: Two Successive Sessions

No.	Source	Destination	Protocol	Src Port	Dst Port	Info
1	192.168.2.43	172.217.10.129	TCP	56380	443	56380 → 443 [SYN] Seq=0 Win=65535 [TCP CHECKSUM INCORRECT]
2	172.217.10.129	192.168.2.43	TCP	443	56380	443 → 56380 [SYN, ACK] Seq=0 Ack=1 Win=60192 Len=0 MS
3	192.168.2.43	172.217.10.129	TLSv1	56380	443	Client Hello
4	192.168.2.43	172.217.10.129	TLSv1	56381	443	Client Hello
5	172.217.10.129	192.168.2.43	TCP	443	56380	443 → 56380 [ACK] Seq=1 Ack=518 Win=61440 Len=0 TSval
6	172.217.10.129	192.168.2.43	TCP	443	56381	443 → 56381 [SYN, ACK] Seq=0 Ack=518 Win=60192 Len=0
7	192.168.2.43	172.217.10.129	TCP	56381	443	56381 → 443 [ACK] Seq=518 Ack=1 Win=131328 [TCP CHECK
8	172.217.10.129	192.168.2.43	TCP	443	56380	[TCP Previous segment not captured] 443 → 56380 [ACK]
9	172.217.10.129	192.168.2.43	TCP	443	56380	443 → 56380 [PSH, ACK] Seq=2837 Ack=518 Win=61440 Len
10	192.168.2.43	172.217.10.129	TCP	56380	443	[TCP Dup ACK 3#1] 56380 → 443 [ACK] Seq=518 Ack=1 Win
11	192.168.2.43	172.217.10.129	TCP	56380	443	[TCP Dup ACK 3#2] 56380 → 443 [ACK] Seq=518 Ack=1 Win
12	172.217.10.129	192.168.2.43	TLSv1	443	56380	[TCP Fast Retransmission] , Server Hello
13	192.168.2.43	172.217.10.129	TCP	56380	443	56380 → 443 [ACK] Seq=518 Ack=3615 Win=129024 [TCP CH
14	192.168.2.43	172.217.10.129	TLSv1	56380	443	Client Key Exchange, Change Cipher Spec, Encrypted Ha

Follow the ports to find the connections...

- Connection 1: Port 56380 to 443
- Connection 2: Port 56381 to 443



Let's examine a TCP Fast Open session. Here is the flow of two successive sessions. We will examine the sessions in more detail in the following slides. The overall flow of this TCP Fast Open traffic is that there is a TCP session between client 192.168.2.43 source port 56380 and server 172.217.10.129 port 443. This session has a standard three-way handshake, with the exception of the combination of data on the ACK that carries a "Client Hello" message, which is used to establish a TLS connection. This is acceptable if the acknowledgement flag is set, and the acknowledgement value is one more than the server's ISN. This efficiency is not related to the TCP Fast Open, as it can occur in a conventional session establishment. The client requests a TCP Fast Open cookie in this session and receives one. We cannot see that in this slide, but we will see it very soon.

The next session starts within milliseconds of the first one, where the client has a new ephemeral port of 56381. It begins with record 4, which is a SYN with a "Client Hello" request that has the issued cookie in it. Again, we cannot see the cookie here, but we will look at that shortly.



To follow along, enter the following command:

wireshark tcp-fastopen.pcap

SYN Fast Open Cookie Request

No.	Source	Destination	Protocol	Src Port	Dst Port
1	192.168.2.43	172.217.10.129	TCP	56380	443
2	172.217.10.129	192.168.2.43	TCP	443	56380
3	192.168.2.43	172.217.10.129	TLS	56380	443

Internet Protocol Version 4, Src: 192.168.2.43, Dst: 172.217.10.129

Transmission Control Protocol, Src Port: 56380, Dst Port: 443,

Source Port: 56380
Destination Port: 443

Sequence number: 0 (relative sequence number)
[Next sequence number: 0 (relative sequence number)]
Acknowledgment number: 0
1011 - Header Length: 44 bytes

Flags: 0x002 (SYN)

Options: (24 bytes), Maximum segment size, No-Operation

- TCP Option - Maximum segment size: 1460 bytes
- TCP Option - No-Operation (NOP)
- TCP Option - Window scale: 6 (multiply by 64)
- TCP Option - No-Operation (NOP)
- TCP Option - No-Operation (NOP)
- TCP Option - Timestamps: TSval 1177404048, TSecr 0
- TCP Option - SYN permitted
- TCP Option - TCP Fast Open
 - Kind: TCP Fast Open Cookie (34)
 - Length: 2
 - Fast Open Cookie Request

[Timestamps]

SANS



tcp-fastopen.pcap

SEC503: Intrusion Detection In-Depth 101

This is the SYN packet on the initial Fast Open session between client IP 192.168.2.43 source port 56380 and server 172.217.10.129 port 443. On the right is an enlarged image of the TCP options—specifically, the Fast Open option with a value of 0x22 or 34. This packet is the client's request for a Fast Open cookie. The request can be distinguished from a response in that the cookie itself will be empty in the request. We know that the cookie is empty because the length field is 2. This indicates that the only things present are the Fast Open option and the length fields.



To follow along, enter the following command:

wireshark tcp-fastopen.pcap

SYN/ACK Fast Open Cookie Returned

No.	Source	Destination	Protocol	Src Port	Dst Port
1	192.168.2.43	172.217.10.129	TCP	56380	443
2	172.217.10.129	192.168.2.43	TCP	443	56380
3	192.168.2.43	172.217.10.129	TLS	56380	443

Internet Protocol Version 4, Src: 172.217.10.129, Dst: 192.168.2.43
 Transmission Control Protocol, Src Port: 443, Dst Port: 56380,

Source Port: 443
 Destination Port: 56380

[TCP Segment Len: 0]
 Sequence number: 0 (relative seq)
 Next sequence number: 0 (relative seq)
 Acknowledgment number: 1 (relative seq)
 1101 - Header Length: 52 bytes

Flags: 0x012 (SYN, ACK)

Options: (32 bytes), Maximum segment size, SACK permitted, Timestamps
 TCP Option - Maximum segment size: 1380 bytes
 TCP Option - SACK permitted
 TCP Option - Timestamps: TSval 3686875425, TSecr 1177404048
 TCP Option - No-Operation (NOP)
 TCP Option - Window scale: 0 (multiply by 256)
 TCP Option - TCP Fast Open
 Kind: TCP Fast Open Cookie (34)
 Length: 10
 Fast Open Cookie: c7615cc24bf34512
 TCP Option - No-Operation (NOP)

SANS



tcp-fastopen.pcap

SEC503: Intrusion Detection In-Depth 102

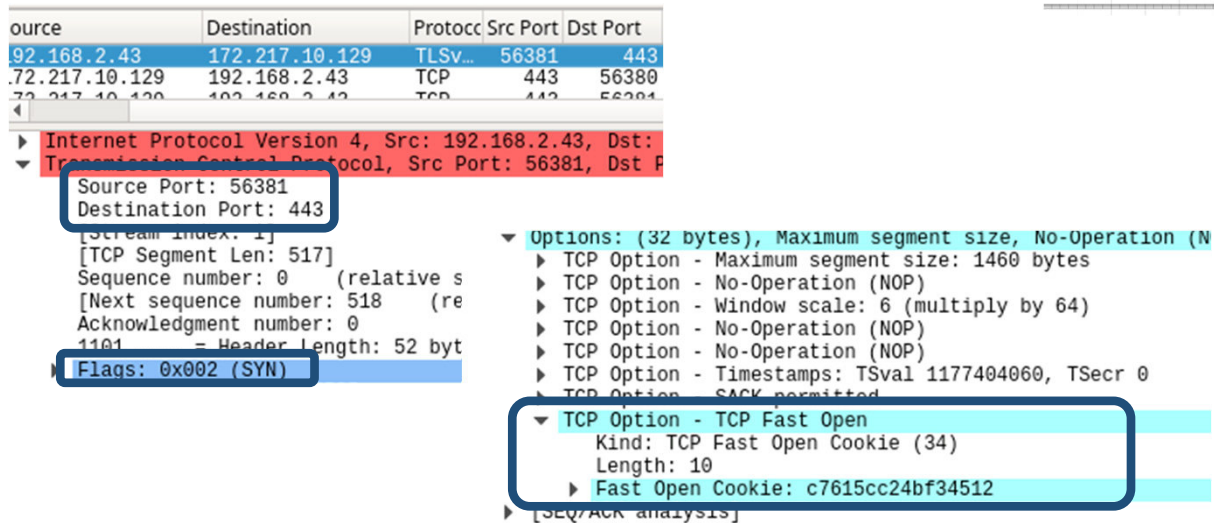
The server responds with a SYN/ACK and a generated cookie if it supports Fast Open. If not, it uses the standard three-way handshake without the cookie. The enlarged image of the TCP options shows that the value generated for the cookie is 0xc7615cc24bf34512.



To follow along, enter the following command:

wireshark tcp-fastopen.pcap

Start New SYN Session with Returned Cookie



Source	Destination	Protocol	Src Port	Dst Port
192.168.2.43	172.217.10.129	TLSv...	56381	443
172.217.10.129	192.168.2.43	TCP	443	56380
172.217.10.129	192.168.2.43	TCP	443	56381

Internet Protocol Version 4, Src: 192.168.2.43, Dst: 172.217.10.129

Transmission Control Protocol, Src Port: 56381, Dst Port: 443

Source Port: 56381
Destination Port: 443

Stream index: 1
[TCP Segment Len: 517]
Sequence number: 0 (relative to stream start)
[Next sequence number: 518 (relative to stream start)]
Acknowledgment number: 0
[Window length: 52 bytes]
Flags: 0x002 (SYN)

Options: (32 bytes), Maximum segment size, No-Operation (NOP), Window scale, Timestamps, SACK permitted

- TCP Option - Maximum segment size: 1460 bytes
- TCP Option - No-Operation (NOP)
- TCP Option - Window scale: 6 (multiply by 64)
- TCP Option - No-Operation (NOP)
- TCP Option - No-Operation (NOP)
- TCP Option - Timestamps: TSval 1177404060, TSecr 0
- TCP Option - SACK permitted
- TCP Option - TCP Fast Open
 - Kind: TCP Fast Open Cookie (34)
 - Length: 10
 - Fast Open Cookie: c7615cc24bf34512

[SEQ/ACK analysis]



This is a subsequent session request from that same client that is opened shortly thereafter. The client is the same, 192.168.2.43, as is the server, 172.217.10.129. This is a new session to destination port 443 because the source port is now 56381. The SYN is set.

The enlarged image of the TCP options shows the Fast Open with the previously generated cookie value 0xc7615cc24bf34512. The result, in this case, is that the client was able to begin the TLS handshake *before* the session initiation completed, saving at least one packet out of the entire conversation.

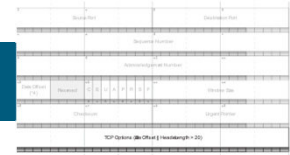
This may not seem like much, but when you are talking about hundreds of millions to billions of connections, the savings begin to add up!



To follow along, enter the following command:

wireshark tcp-fastopen.pcap

TCP Timestamp Option



- Timestamp = Many modern operating systems use:
 - To compute round-trip time for TCP efficiency
 - To auto-tune buffer sizes
 - Protection Against Wrapped Sequence Numbers (PAWS)
- Per RFC 7323:
 - Relative time since last reboot
 - Timestamp tick between 1 ms and 1 second real time

```
172.22.7.133.40655 > 10.125.19.103.80: Flags [S], seq 430116200,
win 5840, mss 1460, sackOK, timestamp 36662154 0, nop, wscale 5],
length 0
```

The TCP timestamp option is defined in RFC 1323, "TCP Extensions for High Performance." Its purpose is to maintain a value representing a notion of the host's current time in a field in the TCP options. Actually, the timestamp field contains both the sender and receiver timestamps. This permits either host to calculate the round-trip time for TCP to efficiently send traffic at a pace where throughput is optimized and packet loss is minimized.

In addition, the timestamp is used for a mechanism known as *Protection Against Wrapped Sequence Numbers (PAWS)*. PAWS attempts to prevent confusion of potentially conflicting TCP sequence numbers if they wrap back to a value of zero or a host crashes and resumes a session upon reboot using the same sequence numbers. When you first hear this, you might think, "How unlikely is it for a packet to "get lost" during a connection, have the sequence numbers wrap around, and then that "lost" packet arrives at just the right moment to corrupt the session?" It turns out this is far more likely than you might think. You see, the sequence number doesn't need to be *exactly* right. It simply needs to fall within the window. What window? The window size! If our connection uses the maximum possible window size of 1,073,725,440 (which is $0xFFFF \ll 14$), the sequence number need only be a number within that billion plus range of values! After all, the IP-port-pair will be correct since it's the same connection. This means that we would automatically corrupt the session, which would be bad.

For most operating systems, the TCP timestamp is a relative value that reflects the number of *clock ticks* since the last reboot. A clock tick for a TCP timestamp is defined in RFC 1323 as anywhere between 1 ms and 1 second of real time. Looking at the tcpdump output on the bottom of the slide, you see the timestamp for host 172.22.7.133 has a value of 36662154, representing the clock ticks since last reboot. The receiver's timestamp has a value of 0 because this is a SYN packet, and no communication has occurred with the receiver yet.

If you want to learn more about TCP timestamps, you can find RFCs 1323 and 7323 at the following sites:
<http://www.ietf.org/rfc/rfc1323.txt>
<http://www.ietf.org/rfc/rfc7323.txt>

Duplicate Acknowledgements – Selective Acknowledgement

```

10.1.3.7.54187 > 10.1.3.55.80: Flags [S], seq 5812322, win 65535,
  options [mss 1460,sackOK,eol]
10.1.3.55.80 > 10.1.3.7.54187: Flags [S.], seq 637843953, ack 5812323, win 5720,
  options [mss 1430,nop,nop,sackOK]
10.1.3.7.54187 > 10.1.3.55.80: Flags [.], ack 1, win 65535
10.1.3.7.54187 > 10.1.3.55.80: Flags [P.], seq 25:29, ack 1, win 65535
10.1.3.55.80 > 10.1.3.7.54187: Flags [.], ack 1, win 5720,
  options [nop,nop,sack 1 {25:29}]
10.1.3.7.54187 > 10.1.3.55.80: Flags [P.], seq 15:25, ack 1, win 65535
10.1.3.55.80 > 10.1.3.7.54187: Flags [.], ack 1, win 5720,
  options [nop,nop,sack 1 {15:29}]
10.1.3.7.54187 > 10.1.3.55.80: Flags [P.], seq 1:15, ack 1, win 65535
10.1.3.55.80 > 10.1.3.7.54187: Flags [.], ack 29, win 5720

```

Let's see how selective acknowledgements (SACK) and duplicate acknowledgements work. We are looking at an exchange between hosts 10.1.3.7 and 10.1.3.55. Both hosts support the TCP selective acknowledgement option. We'll see how this assists in the reporting of duplicate acknowledgements.

You can see that, after the three-way handshake, the sequence numbers switch to relative sequence numbers. We have used this default view to make it easier to see how much data is being transferred. The client's sequence number values have single underlines, whereas the server's acknowledgements of those sequence number values are bolded. The three-way handshake begins with a SYN from 10.1.3.7 with a TCP sequence number of 5812322. The server acknowledges this with an ACK value of 5812323. The third packet shows the completion of the three-way handshake and the switch to relative sequence numbers.

Next, the client sends out-of-order sequence numbers for bytes 25 through 29. The server is expecting to see the first bytes in the session, but these have not appeared yet. Therefore, the server reports back in record 5 using the acknowledgement value that the next expected sequence number is 1, but it also reports by virtue of the selective acknowledgement TCP option that sequence numbers 25–29 were received.

The client sends more out-of-order sequence numbers, 15–25, in record 6. Again, the server responds with an acknowledgement of 1, indicating that this is the last byte that it knows about. It also updates the selective acknowledgement range to 15–29 as received sequence numbers. Finally, in line 8, the client fills in the gap of 1–15 to complete the range of 1–29. The server acknowledges this with an acknowledgement value of 29.

Along the way, you likely noticed things like this: `sack 1 {25:29}`. This indicates that the host is selectively acknowledging that bytes from offset 25 through 29 have arrived out of order and are being buffered in one chunk and will be added when the earlier data arrives. You can see that this chunk expands when the second chunk arrives.

Fingerprinting with TCP Options

The image shows a network packet capture with two packets. The first packet is from 192.168.11.59 to 192.168.11.21.22, showing TCP options: MSS 1460, NOP, Window Scale 8, and SACKOK. The second packet is from 192.168.11.21 to 192.168.11.12.22, showing TCP options: MSS 1460, SACKOK, Timestamp 151137911, and Window Scale 7.

Windows 10 TCP Options

```
192.168.11.59.51262 > 192.168.11.21.22: Flags [S], seq
3093033885, win 8192, options [mss 1460,nop,wscale
8,nop,nop,sackOK]
```

Linux 4.2 Kernel Options

```
192.168.11.21.51226 > 192.168.11.12.22: Flags [S], seq
825630900, win 29200, options [mss 1460,sackOK,TS val
151137911 ecr 0,nop,wscale 7]
```

What do you support, what values do you use, and in which order do the options appear?

SANS



tcp-options.pcap

SEC503: Intrusion Detection In-Depth 106

One of the ways that nmap fingerprints an operating system is by sending different TCP options to the host to see how it responds. Not all operating systems support all the TCP options, so this assists in operating system identification. Also, different operating systems store the TCP options in a unique order in the TCP header. As Fyodor, nmap's author, says, this is a "gold mine" of information for classification.

The first packet on the slide is a SYN from 192.168.11.59, a Windows 10 host. It supports the MSS, the window scale of 8, and the selective acknowledgement. The second packet is a SYN from 192.168.11.21, a host running a Linux 4.2 kernel. It supports the MSS, the selective acknowledgement, the timestamp, and a window scale of 7.

Not only are the TCP options supported for these two hosts different, but the order that they appear in the TCP header is too. The position of the NOP padding can be different among operating systems, although both these operating systems place the pad byte(s) before an option to make it fall on a 4-byte boundary, as required.

The length of TCP options used by the Windows host are 4 bytes for the MSS, 3 bytes for the window scale, and 2 bytes for the selective acknowledgement for a total of 9 bytes, requiring 3 additional bytes of padding. Each option that falls short of a 4-byte boundary is pre-padded with the necessary number of NOPs. The Linux host pre-pads as well. It has a 4-byte MSS, a 2-byte selective acknowledgement, a 10-byte timestamp, and a 3-byte window scale. The 2-byte selective acknowledgement plus the 10-byte timestamp fall on a 4-byte boundary, leaving the 3-byte window scale missing a byte that is supplied by NOP, which precedes it.



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

```
tcpdump -r tcp-options.pcap -nt
```

Section 14.2
Packet Protocols as
a Second Language

- 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

Prerequisites: Networking 101, Fundamentals of Intrusion Detection

- Wireshark Display Filters
- Writing tcpdump Filters
- TCP
- UDP
- ICMP
- IPv6

Fundamentals of Traffic Analysis: Part II

TCP Theory

TCP Header Fields

TCP Stimulus and Response

This page intentionally left blank.

TCP Stimulus: Response

- Using TCP as an example protocol, examine various situations for expected responses.
- Attempt a port 80 connection to the following:
 - A destination host listening on port 80
 - A destination host not listening on port 80
 - A destination host that doesn't exist
 - A destination host whose port 80 connection is blocked by a router that responds with an ICMP error message
 - A destination host whose port 80 connection is blocked by a router that doesn't respond

Let's look at responses to an attempted port 80 connection under varying conditions. In the following slides, you see some of the varied and expected responses to the identical stimulus. Obviously, this is not an exhaustive list of all conditions that you are likely to encounter. These conditions have been selected for illustration because they are some of the most common.

Listening HTTP Server

192.168.122.1 attempts HTTP connection to 192.168.122.129

STIMULUS

192.168.122.1.35049 > 192.168.122.129.80: Flags [S], seq 1856958874, win 63, options [mss 1400,wscale 0,sackOK,TS val 4294967295 ecr 0,eol], length 0

192.168.11.129 listens on port 80 and access is permitted:

RESPONSE

192.168.122.129.80 > 192.168.122.1.35049: Flags [S.], seq 3819046312, ack 1856958875, win 5792, options [mss 1460,sackOK,TS val 190406817 ecr 4294967295,nop,wscale 5], length 0



What is the expected response when client host 192.168.122.1 attempts to connect to port 80 on listening server 192.168.122.129? We've already discussed the concept of the three-way handshake for TCP session establishment. If you remember, the first part of the process requires that the client initiates a TCP connection with the SYN flag set to the server; this signals the desire to connect. In this slide, you see 192.168.122.1 issue such a SYN connection request to 192.168.122.129 to connect to port 80.

Now, if 192.168.122.129 listens on port 80 and access is permitted, and there are no other impediments, you see the expected response from 192.168.122.129 with a SYN/ACK flag combination.



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

tcpdump -r tcp-options.pcap -nt

Host Not Listening on Port 80

STIMULUS

```
192.168.122.1.35798 > 192.168.122.129.80: Flags [S], seq 1297570608, win 5840,
options [mss 1460,sackOK,TS val 253401491 ecr 0,nop,wscale 6], length 0
```

192.168.122.129 doesn't listen on port 80

RESPONSE

```
192.168.122.129.80 > 192.168.122.1.35798: Flags [R.], seq 0, ack 1297570609,
win 0, length 0
```

Notice that this is an RST ACK; by RFC, you must respond to a SYN with an ACK.

SANS



rst.pcap

SEC503: Intrusion Detection In-Depth 110

You see the response from an attempted connection to 192.168.122.129, but this time 192.168.122.129 doesn't listen on port 80. The expected response is an RST/ACK that is an abrupt termination of the connection. Unlike the FIN, no response is expected.



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

```
tcpdump -r rst.pcap -nt
```

Destination Host Doesn't Exist

STIMULUS

```
192.168.122.44.36476 > 192.168.122.129.80: Flags [S], seq 271616355, win 5840,
options [mss 1460,sackOK,TS val 254138841 ecr 0,nop,wscale 6], length 0
```

192.168.122.129 doesn't exist:

RESPONSE

```
192.168.122.1 > 192.168.122.44: ICMP host 192.168.122.129 unreachable, length
68
```

This could also result in a
"network unreachable."

SANS



icmp-hostunreach.pcap

SEC503: Intrusion Detection In-Depth 111

What happens if 192.168.122.44 attempts a port 80 connection to 192.168.122.129, but 192.168.122.129 doesn't exist? If this is on the same broadcast domain, ARP cannot resolve the IP address, and the failure causes the client to conclude that there is "No route to host."

However, when a router or firewall exists where a host cannot respond, the router or firewall may issue an ICMP error. In this case, the 192.168.122.1 is a router that informs 192.168.122.44 via ICMP that 192.168.122.129 is unreachable.



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

```
tcpdump -r icmp-hostunreach.pcap -nt
```

Port 80 Blocked, Router/Firewall Silenced

STIMULUS

```
07:36:52.849306 IP 192.168.122.1.36495 > 192.168.122.129.80: Flags [S], seq
751270588, win 5840, options [mss 1460,sackOK,TS val 254204054 ecr
0,nop,wscale 6], length 0
```

Router/firewall does not respond to blocked port 80 request:

CONTINUED STIMULI—NO RESPONSE

```
07:36:55.846318 IP 192.168.122.1.36495 > 192.168.122.129.80: Flags [S], seq
751270588, win 5840, options [mss 1460,sackOK,TS val 254204804 ecr
0,nop,wscale 6], length 0
```

```
07:37:01.846950 IP 192.168.122.1.36495 > 192.168.122.129.80: Flags [S], seq
751270588, win 5840, options [mss 1460,sackOK,TS val 254206304 ecr
0,nop,wscale 6], length 0
```

SANS



router-silence.pcap

SEC503: Intrusion Detection In-Depth 112

Now you see another condition that may be encountered where port 80 is blocked; however, no ICMP message is issued. Because there is no ICMP error message to inform 192.168.122.1 that something is amiss, it continues to send retries to connect. The number of retries and the time intervals between each are designated by the operating system of the host sending the retries. Finally, the host 192.168.122.1 gives up on the connection after it has exhausted the maximum number of retries.



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

```
tcpdump -r router-silence.pcap -nt
```

Wrap-Up of TCP

- Requires session establishment to start stream
- Flags used to designate current state
- Session typically consists of multiple packets carrying data
- Overhead required for reliability:
 - Sequence numbers
 - Acknowledgement numbers
- Complexity involved may allow for ambiguities to be introduced

In review, you see that TCP is a complex connection-oriented protocol that attempts to ensure reliability. It requires a session establishment, typically accomplished via the three-way handshake, to begin a session. Data is typically exchanged after the three-way handshake. A standard session consists of several packets that carry data. These must be reassembled by the receiving host and an IDS/IPS that attempts to find malicious content.

There is some overhead required to ensure reliability. TCP sequence numbers assign a notion of chronology to each segment. TCP acknowledgement numbers reflect the next expected sequence numbers from the sender. When segments arrive out of order, the receiver issues a duplicate acknowledgement number to inform the sender of the gap in sequence numbers. When the sender does not receive an expected acknowledgement number within a given time, it resends the segment(s).

With reliability comes complexity. Complexity offers more opportunities to introduce ambiguities. These ambiguities may lead to evasions. At issue is a potential mismatch between the way the IDS/IPS and the end host evaluate the traffic. If the evaluation is not identical, an evasion is possible. More on this at the end of Book 4.

TCP Exercises

Workbook exercises "TCP"

This page intentionally left blank.

Session 14.2 Packet Protocols as a Second Language

• 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 Protocols as a Second Language: Fundamentals of Traffic Analysis

- Wireshark Display Filters
- Writing tcpdump Filters
- TCP
- UDP
- ICMP
- IPv6

Fundamentals of Traffic Analysis: Part II

UDP Theory and Header Fields

UDP Stimulus and Response

Compared to TCP, *User Datagram Protocol (UDP)* is a simple protocol. It is often explained as a "send and pray" protocol because there is no guarantee of reliability. It has a simple header of four fields. We cover a couple of examples of sending UDP packets under different circumstances to understand the expected responses.

UDP

- Lightweight protocol
- Supports unicast, broadcast, multicast addresses
- Low overhead
- No built-in reliability at transport layer
- No message ordering
- No flow control
- Encapsulated in IP header (protocol/next header 17)
- IPv6 changes:
 - IPv6 jumbogram change in UDP length
 - UDP checksum required

UDP is considered a lightweight transport protocol because it is simple and makes no guarantee of reliable delivery. Unlike TCP, which supports only single-host-to-single-host (unicast) communications, UDP can deliver traffic to one or more hosts. It has low overhead and can be fast because it has a standard 8-byte header that carries vital information such as source, destination port, UDP checksum, and a length that reflects the number of payload and UDP header bytes.

There is no reliability at the UDP layer, but applications can be written so that they build in reliability. The application itself, and not UDP, will be responsible for ensuring any type of reliability needed. As we witnessed, TCP had a notion of order via TCP sequence numbers for a stream of data. UDP has no such notion of order. And there is no coordination between the sender and receiver for the most efficient amount of data to be sent.

Like TCP, there must be some way to indicate that a packet is an IPv6 jumbogram. Because the IP length field cannot represent a value greater than 65535, another field must be used. A UDP length of 0 designates that the packet is a jumbogram. The UDP length field found in the UDP header reflects the number of bytes in the UDP header plus payload. A value of 0 should never be seen in an IPv4 or IPv6 packet as a valid length; therefore, it is a good jumbogram indicator.

Many “Transactional” Protocols are UDP

Transactional = single request results in single response

- DNS
- NTP
- SNMP
- DHCP

Some leverage UDP for speed (application sequences data)

- NFS (can also run over TCP)
- Various tunneling protocols

Since UDP has no session setup requirements, it is ideal for applications that are more “transactional.” By this, we mean that the protocol typically has a single request that corresponds to a single response. Here are some examples:

- *Domain Name System (DNS)*, which is required to associate the correct hostname and IP address. UDP is an ideal transport for DNS since DNS queries and responses need to be as quick as possible. There are provisions in DNS to use TCP as the transport protocol for larger exchanges such as zone transfers. And DNSSEC is an add-on remediation to include security in DNS at the application layer to accommodate some of the issues that are engendered by using UDP as the transport protocol in the first place.
- *Network Time Protocol (NTP)* is used to synchronize the time on our systems. This is obviously important for logging and scheduling, but less obviously, it is critical to most modern authentication systems.
- *Simple Network Management Protocol (SNMP)* maintains information about network devices. This is useful for diagnosing issues and monitoring network assets through short, simple queries.
- *Dynamic Host Configuration Protocol (DHCP)* is used to assign IP addresses, gateways, and DNS server information. A corruption of this process by altering UDP data can allow an attacker to create a machine-in-the-middle situation, essentially routing all traffic through her computer.

Other protocols use UDP to leverage the lower overhead. Overhead is reduced not only by the smaller, static sized header. The overhead is also dramatically reduced by eliminating the requirement for ACK packets, each of which takes up time on the network.

One example is the *Network File System (NFS)*, which primarily uses UDP. Running over UDP, the NFS protocol is solely responsible for correctly ordering the data transferred and detection and retransmission of missing data. NFS can also be run over TCP.

Other examples include various tunneling protocols. For example, OpenVPN will tunnel across networks using UDP port 1194 in a very efficient way. While this is the preferred operation for OpenVPN (and other tunneling protocols over other ports), some hotels and other networks may not permit these unusual UDP ports, so TCP port 443 (TLS) might be used as the transport.

Sample UDP Packet

No.	Time	Source	Destination	Protocol	Source port	Destination port
1	0.000000	10.3.8.4	10.3.8.108	DNS	domain	51274

Ethernet II, Src: vmware_96:00:18 (00:50:56:96:00:18), Dst: Dell_85:1C:6C (00:20:09:85:1C:6C) Internet Protocol, Src: 10.3.8.4 (10.3.8.4), Dst: 10.3.8.108 (10.3.8.108) User Datagram Protocol, Src Port: domain (53), Dst Port: 51274 (51274) Source port: domain (53) Destination port: 51274 (51274) Length: 89 Checksum: 0xdc20 [correct] Domain Name System (response)
--



There actually isn't much to a UDP header. The one shown here in Wireshark reflects a DNS response. There are source and destination ports. There is a length that includes the header size of 8 bytes and the data that follows. Like TCP, there is a checksum that also includes a pseudo-header in the algorithm to make sure the packet is delivered to the correct host.



To display the output, enter the following command:

wireshark udp.pcap

UDP Length

- 16-bit field
- Represents length of header and UDP data
- Should have a minimum length of 8
- Value of 0 indicates the packet is a jumbogram

The UDP length is the number of bytes found in the UDP header plus the number of bytes found in the payload. The UDP header is 8 bytes. Remember that a 0 in this field is the indicator that the packet is a jumbogram. It seems strange that the fixed-length 8-byte header size is included in the length.

Checksum

- Validates UDP header and data do not get corrupted in transit
- Same formula as IP and TCP checksums
- Optional in IPv4, mandated in IPv6
- Includes pseudo-header

Like TCP, the UDP header has a checksum field. This checksum value is computed by taking the standard 16-bit 1's complement formula for the values in the UDP header and data, as well as the pseudo-header fields, and placing the result in the UDP checksum field. The receiving host applies this same formula over the same fields and compares its results with the value in the UDP header. The packet is passed to the application layer if they match; otherwise, it is silently discarded. The pseudo-header includes the same fields as the TCP pseudo-header. Its purpose is to make sure the packet arrives at the correct host.

The UDP checksum is optional in IPv4, though almost always used. It is mandated in IPv6 because there is no IP header checksum requiring the transport layer to include a checksum.

Traceroute

- Figures out hops between source and destination
- Sends three UDP packets to each hop
- Uses incrementing TTL values to advance to next hop
- Discovers next-hop location and timing via ICMP time exceeded in transit error

The UNIX traceroute command uses a combination of UDP and ICMP to discover the path that a packet takes from source to destination. This traceroute program is similar in function to the Windows's tracert; however, instead of using ICMP exclusively to discover the routers traversed to reach the destination host, traceroute uses UDP too.

Traceroute begins by sending three UDP packets with a destination port of 33434–33534 to the destination host. Three packets are sent to each hop for redundancy in case there is packet loss. Each of these three packets has a beginning TTL of 1.

The first hop router returns an ICMP **time exceeded in transit** message because the decremented TTL value becomes 0. The sending host records the first hop using the source IP address that returned the ICMP error message. Each subsequent hop from source to destination generates three new UDP packets with incrementing UDP ports and a TTL value of one more than the TTL value found in the set of the three UDP packets from the previous iteration. Each of the encountered routers returns the ICMP **time exceeded in transit** error, and traceroute records the router's IP address as the next hop.

When and how does this all stop? Eventually, the UDP packets reach the destination host. Most likely, the destination host does not listen on those high-numbered UDP ports, so it issues an ICMP **port unreachable** message.

Traceroute Output

```
traceroute 172.16.0.2
```

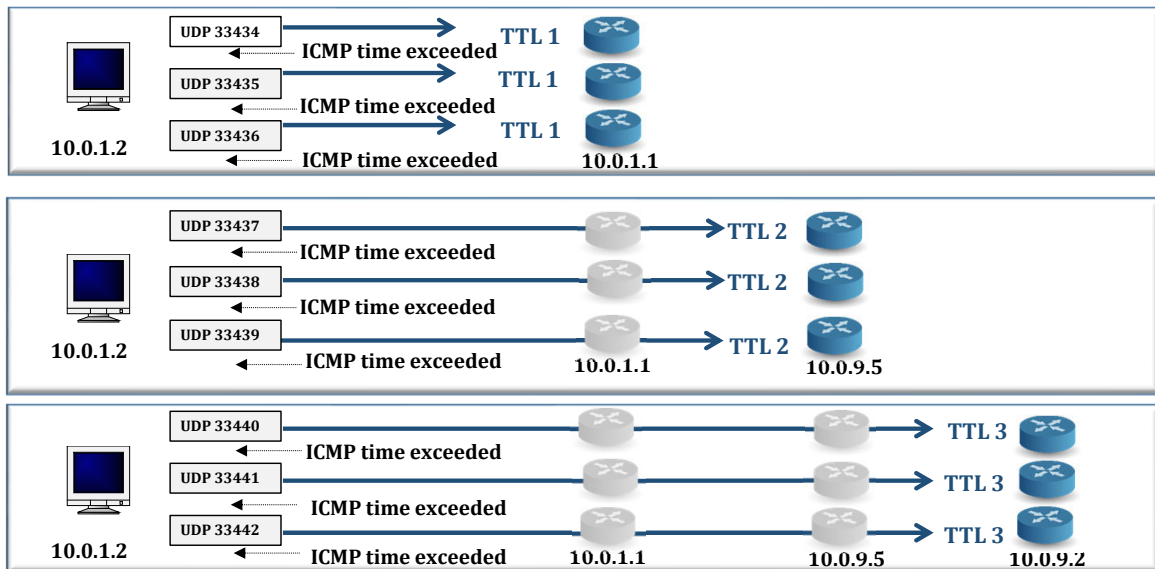
1	DD-WRT 10.0.1.1	0.293 ms	0.438 ms	0.485 ms
2	10.0.9.5	1.861 ms	1.897 ms	1.939 ms
3	10.0.9.2	27.656 ms	27.705 ms	28.448 ms
4	10.0.2.1	15.626 ms	15.678 ms	15.650 ms
5	10.0.2.2	16.072 ms	16.138 ms	16.182 ms

Let's say that host 10.0.1.2 performs a traceroute to 172.16.0.2. The first set of three UDP packets sent to the destination address of 172.16.0.2 contains a TTL value of 1. Router 10.0.1.1 returns an ICMP **time exceeded in transit** message. The TTL value of 1 is placed in the IP header in the next set of three UDP packets from 10.0.1.2, where we assume that they pass through 10.0.1.1 to reach 10.0.9.5. It is possible that this set of packets will take a different route than the first set; however, this example shows all packets using the same route.

This process continues through hops 10.0.9.2, 10.0.2.1, and 10.0.2.2, where each set of three UDP packets gets an incremented TTL value per hop and each individual UDP packet gets an incrementing destination port. The process is repeated until the packets reach the destination IP, which returns an ICMP **port unreachable** message.

Here is the type of output you might see. Traceroute reports each hop as it is returned in an ICMP **time exceeded in transit** message, along with the round-trip time of each.

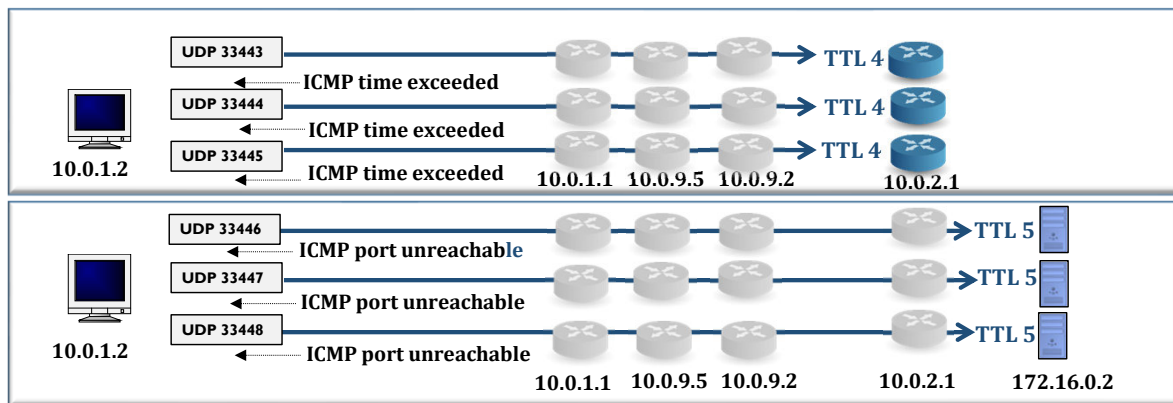
Traceroute Part I



Perhaps it might be easier to understand traceroute by looking at a pictorial representation of what is happening. At the top, you see the initial discovery of the first hop. Three UDP packets are sent, each with an incrementing UDP port of 33434, 33435, and 33436. Each has a TTL value of 1. And 10.0.1.1 as the first hop returns an ICMP **time exceeded in transit** error message for each.

This is repeated for the second hop of 10.0.9.5, except each UDP packet has a destination port of 33437, 33438, and 33439, and each has a TTL value of 2. This is repeated for the next hop of 10.0.9.2, except each UDP packet has a destination port of 33440, 33441, and 33442, and each has a TTL value of 3.

Traceroute Part 2



The fourth hop of 10.0.2.1 is reached using a TTL value of 4 on each of the ports 33443, 33444, and 33435. The TTL is incremented to 5, allowing the UDP packet to reach the destination host 172.16.0.2. It does not listen on any of the ports 33446, 33447, or 33448, so an ICMP **port unreachable** message is returned. After the traceroute program receives this ICMP error, it knows that it has reached the target host.

Section 14.2
Packet/Networks as
a Second Language

- 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

Prerequisite: Section 14.1, Packet/Networks as a Second Language

Fundamentals of Traffic Analysis: Part II

UDP Theory and Header Fields

UDP Stimulus and Response

- Wireshark Display Filters
- Writing tcpdump Filters
- TCP
- UDP
- ICMP
- IPv6

This page intentionally left blank.

UDP Stimulus: Response

- Using UDP as an example protocol, examine two situations for expected responses.
- Attempt to issue a DNS name lookup to the following:
 - A destination host listening on the domain port
 - A destination host not listening on the domain port

Let's look at how UDP responds to incoming packets. We use a DNS query to examine how UDP behaves. Specifically, we look at a listening domain port and a non-listening one. The other conditions examined for TCP, such as a host that doesn't exist or the domain port blocked at the router, elicit a similar response, so we won't bother to discuss them.

Host Not Listening Port 53

No.	Time	Source	Destination	Protocol	Src Port	Dest Port	Info
1	0.000000	192.168.122.1	192.168.122.129	DNS	45756	53	Standard query 0x8b6b A www.sans.org
2	0.000505	192.168.122.129	192.168.122.1	ICMP	45756	53	Destination unreachable (Port unreachable)

Frame 2: 100 bytes on wire (800 bits), 100 bytes captured (800 bits) on interface 0	
Ethernet II, Src: 00:0c:29:f1:b9:a3 (00:0c:29:f1:b9:a3), Dst: 00:50:56:c0:00:01 (00:50:56:c0:00:01)	
Internet Protocol Version 4, Src: 192.168.122.129 (192.168.122.129), Dst: 192.168.122.1 (192.168.122.1)	
Internet Control Message Protocol	
Type: 3 (Destination unreachable)	
Code: 3 (Port unreachable)	
Checksum: 0x7308 [correct]	
Internet Protocol Version 4, Src: 192.168.122.1 (192.168.122.1), Dst: 192.168.122.129 (192.168.122.129)	
Version: 4	
Header length: 20 bytes	
Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00: Not-ECT (Not ECN-Capable Transport))	
Total Length: 58	
Identification: 0x8f29 (36649)	
Flags: 0x00	
Fragment offset: 0	
Time to live: 64	
Protocol: UDP (17)	
Header checksum: 0x75b6 [in ICMP error packet]	
Source: 192.168.122.1 (192.168.122.1)	
Destination: 192.168.122.129 (192.168.122.129)	
[Source GeoIP: Unknown]	
[Destination GeoIP: Unknown]	
User Datagram Protocol, Src Port: 45756 (45756), Dst Port: 53 (53)	
Domain Name System (query)	

SANS



dns-portunreachable.pcap

SEC503: Intrusion Detection In-Depth 127

Here we can see the response that is generated when a UDP packet is sent to a non-listening (or closed) port. In this case, 192.168.122.129 responds that port 53 is unreachable.

However, don't automatically assume that a lack of an ICMP port unreachable message means a listening UDP port. There may be some outbound filtering that prevents the ICMP message from leaving the network.



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

wireshark dns-portunreachable.pcap

Section 14.2 Packet Networks as a Second Language

• 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 Networks as a Second Language: Fundamentals to the real-world packet network

- Wireshark Display Filters
- Writing tcpdump Filters
- TCP
- UDP
- ICMP
- IPv6

Fundamentals of Traffic Analysis: Part II

ICMP Theory and Header Fields

The *Internet Control Message Protocol (ICMP)* is a vital part of IP and plays a vital role in delivering messages about error conditions it finds, as well as delivering simple requests and replies.

ICMP is a lightweight protocol originally created for network troubleshooting. ICMP, like IP and UDP, has no built-in reliability. ICMP packets can be lost or dropped as well with no accountability.

ICMP

- Encapsulated in IPv4 header (protocol = 1)
- Reports about non-transient problems:
 - To IP (fragmentation required, DF set)
 - To transport layer (port unreachable)
- Also used for simple informational exchanges
- Error messages contain embedded data regarding the "offending" packet
- Changes for IPv6
 - ICMPv6 is protocol 58—see the IPv6 section for more information

ICMP was created to report about problems that are not transient or not likely to go away soon. For instance, if a packet is sent and becomes corrupted in transit, this error is not reported by ICMP. Data corruption is considered to be transient and likely disappears the next time the packet is sent, assuming that there are no permanent software/hardware problems creating it.

A problem that is deemed non-transient is something like a situation where a large packet requires fragmentation to be transmitted in full, but the Don't Fragment flag is set. In other words, the packet needs to travel over a network segment with an MTU that is smaller than the packet size. This is not a problem unless the packet has the Don't Fragment flag set. As the name implies, the packet will not be fragmented if the Don't Fragment flag is set. This is reported back to the sending host via an ICMP message.

Another example of a non-transient problem at the transport layer is a UDP packet sent to a non-listening port. This problem is not likely to go away, no matter how many UDP packets are sent. ICMP is enlisted to inform the sending host that the UDP port is not reachable.

ICMP messages are used for simple informational exchanges as well. The most well-known ICMP application is certainly the ICMP echo / echo reply, or ping. Ping is used to find whether a given network host is reachable.

An ICMP error message is supposed to contain, at least, the IP header and 64 bits of the packet that caused the error. The offending host uses the returned ICMP error with the embedded packet to associate it with the error causing packet.

ICMPv6 has greatly extended the use of ICMP to include network discovery via Neighbor Discovery Protocol (NDP). NDP messages can be considered informational exchanges, too.

ICMP Header

Type	Code	Checksum	
< - - use depends on type - - >			

Similar to the UDP header, the ICMP header is a fixed 8 bytes in length. The first 4 bytes of the ICMP header consist of a 1-byte type field, followed by a 1-byte code field, and then a 2-byte checksum field.

The next 4 bytes of the header depend on the value in the ICMP type field. Some ICMP types utilize these 4 bytes, like echo / echo reply. Others simply ignore them, and the fields go unused. They are still valid fields and will be all zeros if they are not used.

ICMP Type

- Type: zero byte offset of ICMP header
 - Determines the format for the rest of the ICMP message
- Several types have been deprecated:
 - RFC6633, RFC6918

Type	Name	Type	Name	Type	Name	Type	Name
0	Echo Reply	10	Router Solicitation	20-29	Reserved (for Robustness Experiment)	39	~~SKIP~~
1	Unassigned	11	Time Exceeded	30	~~Traceroute~~	40	Photuris
2	Unassigned	12	Parameter Problem	31	~~Datagram Conversion Error~~	41	Experimental mobility protocols
3	Destination Unreachable	13	Timestamp	32	~~Mobile Host Redirect~~	42	Extended Echo Request
4	~~Source Quench~~	14	Timestamp Reply	33	~~IPv6 Where Are You~~	43	Extended Echo Reply
5	Redirect	15	~~Information Request~~	34	~~IPv6 I Am Here~~	44-252	Unassigned
6	~~Alternate Host Address~~	16	~~Information Reply~~	35	~~Mobile Registration Request~~	253	RFC3692-style Experiment 1
7	Unassigned	17	~~Address Mask Request~~	36	~~Mobile Registration Reply~~	254	RFC3692-style Experiment 2
8	Echo	18	~~Address Mask Reply~~	37	~~Domain Name Request~~	255	Reserved
9	Router Advertisement	19	Reserved (for Security)	38	~~Domain Name Reply~~		

The zero byte offset of the ICMP header contains the type field. The value in the type field determines the category of ICMP message and the format for the rest of the ICMP message.

As a 1-byte field, there are 256 possible options, but many of the values are unassigned or have been deprecated. You can get a full listing of the types here:

<https://www.iana.org/assignments/icmp-parameters/icmp-parameters.xhtml>

ICMP Code

- Code: first byte offset of ICMP header
- ICMP types will have one or more different codes.

Type	Name	Code	Description
8	Echo	0	-

0	Echo Reply	0	-
---	------------	---	---

11	Time Exceeded	0	Time Exceeded in Transit
		1	Fragment Reassembly

Type	Name	Code	Description
3	Destination Unreachable	0	Network Unreachable
		1	Host Unreachable
		2	Protocol Unreachable
		3	Port Unreachable
		4	Frag needed, DF set
		5	Source Route Failed
		6	Destination Network Unknown
		7	Destination Host Unknown
		8	Source Host Isolated
		...	And more

The ICMP code is in the first byte offset of the ICMP header. The code is a subcategory value for the ICMP message based on the specific type. Some ICMP types have only one code option, like timestamp and echo. Other ICMP types have many code options. The destination unreachable type (3) has 15 available codes.

A listing of all the codes associated with each type can be found here:

<https://www.iana.org/assignments/icmp-parameters/icmp-parameters.xhtml>

Checksums

- IPv4 pseudo-header not included in checksum computation
- IPv6 pseudo-header included in checksum computation

ICMP uses the ICMP header and payload to compute the checksum. The pseudo-header, including fields from the IPv4 header, wasn't deemed necessary for the ICMP checksum because the IPv4 header had its own checksum as a cross-check for the ICMP checksum.

With ICMPv6, the IPv6 update to ICMP, the checksum computation includes the pseudo-header like with TCP and UDP. There is no IPv6 checksum necessitating that the pseudo-header is included with ICMPv6.

ICMP Echo / Echo Reply

- ICMP Echo (type 8, code 0) / ICMP Echo Reply (type 0, code 0)
 - Utilize the four additional bytes of the ICMP header
- 4th and 5th byte offset is the Identifier
- 6th and 7th byte offset is the Sequence Number

Type	Code	Checksum	
Identifier		Sequence Number	

Let's look at how the ICMP type impacts the rest of the ICMP message.

If you examine an ICMP echo message, you will find two additional fields in the ICMP header. The ICMP identifier field is found in bytes 4 and 5 offset, and the ICMP sequence number field is found in bytes 6 and 7 offset.

These fields are used in an echo / echo reply pair to uniquely identify requests, pair them with responses, and keep track of the replies.

ICMP ID / Sequence Number

```
$ tcpdump -ntr icmp-ping.pcap 'host 192.168.238.10'

IP 192.168.238.10 > 192.168.238.2: ICMP echo request, id 47, seq 1, length 64
IP 192.168.238.2 > 192.168.238.10: ICMP echo reply, id 47, seq 1, length 64
IP 192.168.238.10 > 192.168.238.2: ICMP echo request, id 47, seq 2, length 64
IP 192.168.238.2 > 192.168.238.10: ICMP echo reply, id 47, seq 2, length 64
IP 192.168.238.10 > 192.168.238.2: ICMP echo request, id 47, seq 3, length 64
IP 192.168.238.2 > 192.168.238.10: ICMP echo reply, id 47, seq 3, length 64
IP 192.168.238.10 > 69.21.118.171: ICMP echo request, id 48, seq 1, length 64
IP 69.21.118.171 > 192.168.238.10: ICMP echo reply, id 48, seq 1, length 64
IP 192.168.238.10 > 192.168.238.2: ICMP echo request, id 47, seq 4, length 64
IP 192.168.238.2 > 192.168.238.10: ICMP echo reply, id 47, seq 4, length 64
IP 192.168.238.10 > 69.21.118.171: ICMP echo request, id 48, seq 2, length 64
IP 69.21.118.171 > 192.168.238.10: ICMP echo reply, id 48, seq 2, length 64
IP 192.168.238.10 > 192.168.238.2: ICMP echo request, id 47, seq 5, length 64
IP 192.168.238.2 > 192.168.238.10: ICMP echo reply, id 47, seq 5, length 64
```



The ICMP identifier (id) is used to track echo / echo reply communications sent between two hosts. Most hosts pick a new ICMP id for each new echo / echo reply conversation.

The ICMP sequence number tracks each individual echo message that was sent during that conversation. For each new echo message sent, the sequence number increases by one. This allows the host to match the correct echo reply with the echo that was sent.

On the slide, 192.168.238.10 sends an echo to 192.168.238.2 with an id of 47. All the echo messages sent as part of that conversation will use the id of 47. Each individual echo message sent has a unique sequence number starting at 1.

Host 192.168.238.10 starts another conversation with 69.21.118.171. Notice the identifier has changed to 48, and the sequence numbers start back at number one.

ICMP ID / Seq Num: Windows 10 (I)

```
$ tcpdump -ntr icmp-ping.pcap 'host 192.168.238.13'
```

```
IP 192.168.238.13 > 192.168.238.2: ICMP echo request, id 1, seq 5, length 40
IP 192.168.238.2 > 192.168.238.13: ICMP echo reply, id 1, seq 5, length 40
IP 192.168.238.13 > 192.168.238.2: ICMP echo request, id 1, seq 6, length 40
IP 192.168.238.2 > 192.168.238.13: ICMP echo reply, id 1, seq 6, length 40
IP 192.168.238.13 > 69.21.118.171: ICMP echo request, id 1, seq 7, length 40
IP 69.21.118.171 > 192.168.238.13: ICMP echo reply, id 1, seq 7, length 40
IP 192.168.238.13 > 192.168.238.2: ICMP echo request, id 1, seq 8, length 40
IP 192.168.238.2 > 192.168.238.13: ICMP echo reply, id 1, seq 8, length 40
IP 192.168.238.13 > 69.21.118.171: ICMP echo request, id 1, seq 9, length 40
IP 69.21.118.171 > 192.168.238.13: ICMP echo reply, id 1, seq 9, length 40
IP 192.168.238.13 > 192.168.238.2: ICMP echo request, id 1, seq 10, length 40
IP 192.168.238.2 > 192.168.238.13: ICMP echo reply, id 1, seq 10, length 40
IP 192.168.238.13 > 69.21.118.171: ICMP echo request, id 1, seq 11, length 40
IP 69.21.118.171 > 192.168.238.13: ICMP echo reply, id 1, seq 11, length 40
```



Windows 10 takes a slightly different approach with echo / echo reply messages. All Windows 10 echo messages have an id of 1, and each new echo sent increases the sequence number by 1.

Notice on the slide, host 192.168.238.13 is sending echo messages to two different destination IPs. The sequence numbers are sequentially incrementing with each new message regardless of the destination host.

ICMP ID / Seq Num: Windows 10 (2)

```
$ tcpdump -nr icmp-ping.pcap 'host 192.168.238.13'
```

```
18:07:39.852058 IP 192.168.238.13 > 192.168.238.2: ICMP echo request, id 1, seq 5, length 40
18:07:39.852522 IP 192.168.238.2 > 192.168.238.13: ICMP echo reply, id 1, seq 5, length 40
18:07:40.875719 IP 192.168.238.13 > 192.168.238.2: ICMP echo request, id 1, seq 6, length 40
18:07:40.876686 IP 192.168.238.2 > 192.168.238.13: ICMP echo reply, id 1, seq 6, length 40
18:07:41.786257 IP 192.168.238.13 > 69.21.118.171: ICMP echo request, id 1, seq 7, length 40
18:07:41.835229 IP 69.21.118.171 > 192.168.238.13: ICMP echo reply, id 1, seq 7, length 40
18:07:41.906923 IP 192.168.238.13 > 192.168.238.2: ICMP echo request, id 1, seq 8, length 40
18:07:41.907716 IP 192.168.238.2 > 192.168.238.13: ICMP echo reply, id 1, seq 8, length 40
18:07:42.829597 IP 192.168.238.13 > 69.21.118.171: ICMP echo request, id 1, seq 9, length 40
18:07:42.877263 IP 69.21.118.171 > 192.168.238.13: ICMP echo reply, id 1, seq 9, length 40
18:07:42.922032 IP 192.168.238.13 > 192.168.238.2: ICMP echo request, id 1, seq 10, length 40
18:07:42.922628 IP 192.168.238.2 > 192.168.238.13: ICMP echo reply, id 1, seq 10, length 40
18:07:43.859048 IP 192.168.238.13 > 69.21.118.171: ICMP echo request, id 1, seq 11, length 40
18:07:43.907129 IP 69.21.118.171 > 192.168.238.13: ICMP echo reply, id 1, seq 11, length 40
18:07:43.939940 IP 192.168.238.13 > 192.168.238.2: ICMP echo request, id 1, seq 12, length 40
18:07:43.940773 IP 192.168.238.2 > 192.168.238.13: ICMP echo reply, id 1, seq 12, length 40
```



Windows 10 follows this pattern even if there are multiple echo / echo reply conversations happening between the same two hosts. This may lead to difficulty in determining how many different echo / echo reply conversations are happening between two hosts.

One option may be to use timestamps. The Windows ping command sends an echo message roughly every second. Multiple echo messages within the same second may indicate multiple pings are being run from that host.

ICMP Echo – Payloads

- Linux Payload

```
$ tcpdump -ntr icmp-ping.pcap 'host 192.168.238.10' -Xc1

IP 192.168.238.10 > 192.168.238.2: ICMP echo request, id 37, seq 1, length 64
 0x0000: 4500 0054 dfdd 4000 4001 fd6c c0a8 ee0a  E..T..@.@..1....
 0x0010: c0a8 ee02 0800 af71 0025 0001 0645 0061  ....q.%.E.a
 0x0020: 0000 0000 82ef 0000 0000 0000 1011 1213  .....
 0x0030: 1415 1617 1819 1a1b 1c1d 1e1f 2021 2223  .....!"#
 0x0040: 2425 2627 2829 2a2b 2c2d 2e2f 3031 3233  $%&'()*+,-./0123
 0x0050: 3435 3637                                     4567
```

- Windows Payload

```
$ tcpdump -ntr icmp-ping.pcap 'host 192.168.238.13' -Xc1

IP 192.168.238.13 > 192.168.238.2: ICMP echo request, id 1, seq 11, length 40
 0x0000: 4500 003c f1d4 0000 8001 eb8a c0a8 ee0d  E..<.....
 0x0010: c0a8 ee02 0800 4d50 0001 000b 6162 6364  ....MP....abcd
 0x0020: 6566 6768 696a 6b6c 6d6e 6f70 7172 7374  efghijklmnopqrst
 0x0030: 7576 7761 6263 6465 6667 6869             uvwabcdefghi
```



The payload for echo messages vary by operating system and even applications running on those operating systems.

The default payload for many Linux systems usually contains a timestamp followed by incrementing hex values starting at 0x10.

Windows uses incrementing characters starting with lowercase "a" through "w" followed by "a" through "i".

It is important to understand the default and normal behaviors of the systems in your environment. This will help find systems or applications that should not be there. For example, if you were entirely running in a Windows environment, it would be suspicious to see ICMP echo messages with the incrementing hex values seen in the Linux payloads.

ICMP Echo – Strange Payloads

Microsoft
Outlook
Autodiscover

```
IP 10.8.0.9 > 40.97.24.50: ICMP echo request, id 1, seq 5665, length 72
0x0000: 4500 005c 1050 0000 0101 5eae 0a08 0009 E..\.P....^....
0x0010: 2861 1832 0800 e196 0001 1621 92cc df5f (a.2.....!..._
0x0020: 10d5 0460 b4f8 f40c acf8 f40c d2b3 e15f ...`.....-
0x0030: 3e45 9b12 e804 0000 68f5 d32d 3200 0000 >E.....h...-2...
0x0040: 3cf5 d32d 38f5 d32d 10f5 d32d 06c9 df5f <...-8...-....-
0x0050: 3e52 756e 6e69 6e67 00f5 d32d >Running...
```

Microsoft
Telemetry
Data

```
IP 10.4.142.24 > 40.97.190.2: ICMP echo request, id 1, seq 352, length 10
0x0000: 4500 001e 65cf 4000 1c01 7a90 0a04 8e18 E...e.@...z....
0x0010: 2861 be02 0800 f69e 0001 0160 0000 0000 (a.....`....
0x0020: 0000 0000 0000 0000 0000 0000 0000 .....
.....
```

Uh, SSH?

```
14:49:17.867067 IP 184.168.221.63 > 92.68.122.132: ICMP echo request, id
61093, seq 1, length 916
0x0000: 4500 03a8 0000 4000 4001 caa4 b8a8 dd3f E.....@.@.....?
0x0010: 5c44 7a84 0800 03ec eea5 0001 d520 0880 \Dz.....
0x0020: 0000 0000 0000 0000 4000 0002 0000 0000 .....@.....
0x0030: 0000 0370 0001 eea5 5353 482d 322e 302d ...p....SSH-2.0-
0x0040: 4f70 656e 5353 485f 352e 3570 3120 4465 OpenSSH_5.5p1.De
0x0050: 6269 616e 2d36 0d0a 0000 034c 0614 c470 bian-6....L...p
```

Malicious applications and command / control (C2) channels may also use ICMP echo / echo reply messages. The last example on the slide is an example of an application tunneling SSH traffic over ICMP echo / echo reply messages. This is probably not something you want to see on your network. But not all strange echo payloads are malicious. Sometimes our vendors decide to use different payloads in their applications as well.

The first example on the slide turned out to be echo messages from Microsoft Outlook but not before generating some concern on the network where this was discovered. It is a bit strange and concerning to see "Running" in the payload of an ICMP echo message generated from a Windows client.

The middle example on the slide was equally concerning from another network. These messages initially looked like C2 beaconing traffic. These packets were also from a Windows client. Notice, there are several odd things about these packets, including only a 2-byte payload and an IP TTL of 30. After investigating and several months of discussions with Microsoft, it was determined this traffic is from their Footprints service sending telemetry data to Microsoft.

ICMP Echo Reply

“To form an echo reply message, the source and destination addresses are simply reversed, the type code changed to 0, and the checksum recomputed.” —RFC792

Does “the type code changed to 0” mean:

- The type *AND* code fields are both changed to zero.
- OR
- The code in the type field is changed to zero.

ICMP echo reply messages should look almost identical to the received echo message. Per the RFC, the payload of the echo reply should be identical to the original echo message. The only exception is if there are timestamps or record route options, they should be updated accordingly. For example, the timestamp in the Linux echo payload should be updated, and the rest of the incrementing hex values should be exactly the same.

The IP and ICMP headers should be nearly identical as well. In the IP header, the source and destination addresses are reversed. The ICMP header will change from a type 8 (echo) to a type 0 (echo reply). The checksums in both headers must also be updated.

Looking a bit more into the ICMP header changes, RFC792 specifically says, "the type code changed to 0." Does this actually mean to update the type and code to zero or simply change code in the type field to zero? Does it really matter? ICMP type 8 and type 0 messages only have one code, zero.

<https://tools.ietf.org/html/rfc792>

ICMP Echo Reply difference

- Scapy (*ssssh, we cover Scapy tomorrow*)

```
>>> linux_host = IP(src="192.168.238.2", dst="192.168.238.10")/ICMP(type=8, code=4)/"I <3 ICMP!"
>>> win_host = IP(src="192.168.238.2", dst="192.168.238.13")/ICMP(type=8, code=4)/"sec503 is fun!!"

>>> send(linux_host)

>>> send(win_host)
```

- Notice the Linux host responds differently than the Windows host.

```
$ sudo tshark -n -i ens33 -Y 'icmp' -T fields -e ip.src -e ip.dst -e icmp.type -e icmp.code -e data.data
Running as user "root" and group "root". This could be dangerous.
Capturing on 'ens33'
192.168.238.2 192.168.238.10 8 4 49203c332049434d5021
192.168.238.10 192.168.238.2 0 4 49203c332049434d5021
192.168.238.2 192.168.238.13 8 4 7365633530332069732066756e2121
192.168.238.13 192.168.238.2 0 0 7365633530332069732066756e2121
```

What happens if we send ICMP type 8, code 4 messages to different operating systems? This type and code combination isn't valid, but most operating systems will still respond with an echo reply. Using Scapy, we can craft and send ICMP type 8, code 4 messages.

Don't worry about the Scapy commands for now. There is a full, fun section on Scapy coming in Section 3.

Linux operating systems seems to have interpreted the RFC as changing the code in the type field to zero and sending everything else back the same. Notice, the Linux response is a type 0, code 4 message.

Windows systems respond differently. They change the type and code both to zero.

While overall innocuous, this does provide another potential way to determine a host operating system. It also highlights how RFCs can be difficult to understand and may be interpreted differently.

ICMP Time Exceeded

- Type 11: Time Exceeded
 - Code 0: Time to Live exceeded in transit
 - Code 1: Fragment reassembly time exceeded
- Error message payloads include information about offending packet

Type	Code	Checksum
	< - - unused - - >	

IP Header plus 64 bits of the Original Packet

Let's look at another ICMP type and how it impacts the ICMP message. The type 11 or time exceeded message is an ICMP error message that has two possible code options.

Code 0, Time to Live exceeded in transit, is used to inform the originator when the IP Time to Live field in a packet is zero and a system is discarding the offending packet. This type and code is sent by the system that is dropping the packet, which may be a router or host system.

The second type 11 code available is a code 1, fragment reassembly time exceeded. This message is sent by a host system that does not receive all the fragments of a fragmented packet before the host's timeout setting. The first fragment must have been received for this message to be sent.

ICMP type 11 messages do not use offsets 4 through 7 in the ICMP header, so they will remain zero.

The payload, being an ICMP error message, should contain the IP header and at least 64 bits of the packet that caused the error. For a type 11, code 1 message, this would be the information from the first fragment.

ICMP Time Exceeded – Windows tracert

Type 11
Code 0

Notice the 2nd IP header in the payload

The Windows tracert command uses a combination of echo, echo reply, and time exceeded messages to discover routers that a packet traverses between a source and a destination.

By default, tracert sends a series of three echo (type 8, code 0) packets with an initial IP TTL value of 1. The first router will decrement the IP TTL to zero and send a time exceeded in transit (type 11, code 0) message back to the sending host. Since the router sent the time exceeded error, the host now knows the IP address of the first router.

Tracert will then send another series of three echo packets, increasing the IP TTL to 2. This time the echo message will move past the first router, and the second router will generate the time exceeded error. This gives tracert the second router's IP address.

This behavior, increasing the TTL and router generating an error, will continue until the destination host receives the echo message. The destination host will respond with an echo reply (type 0, code 0) which will indicate to tracert that it reached its destination.

In the slide, notice there are additional IP and ICMP headers in the details pane. These are the headers from the offending packet, so the sending host can map the error to the correct packet.



ICMP Destination Unreachable (I)

- Type 3: Destination Unreachable
 - Many codes assigned: 1 to 15
- Error message payloads include information about offending packet

Type	Code	Checksum
		
	< - - unused - - >	
		

IP Header plus 64 bits of the Original Packet

Another type of ICMP message is the type 3, or destination unreachable message. Like type 11, type 3 messages do not use offsets 4 through 7 of the ICMP header. Type 3 messages are ICMP error messages, so they will include the IP header and 64 bits of the offending packet in the message payload.

There are many different codes assigned to type 3 messages. Some codes notify when a port or protocol isn't available on the destination host. Other codes are for when hosts or networks are unreachable or unknown.

Type 3, code 4 messages are sent when fragmentation is needed but the do not fragment bit was set. These messages are used to notify a sending host that the packet was too big and needs to be fragmented. They are also used with Path MTU Discovery (PMTUD) to determine the maximum transmission unit on a given path.

For a full list of type 3 codes, see this site:

<https://www.iana.org/assignments/icmp-parameters/icmp-parameters.xhtml#icmp-parameters-codes-3>

ICMP Destination Unreachable (2)

ICMP error messages could indicate recon activities.

```
$ tcpdump -ntr icmp-error.pcap 'icmp[0]=3'
```

```
IP 69.21.118.2 > 64.39.99.30: ICMP 172.23.255.3 protocol 250 unreachable, length 29
IP 69.21.118.75 > 64.39.99.30: ICMP 10.4.193.132 protocol 250 unreachable, length 29
IP 69.21.118.75 > 64.39.99.30: ICMP 10.4.193.132 protocol 251 unreachable, length 29
IP 69.21.118.75 > 64.39.99.30: ICMP 10.4.193.132 protocol 252 unreachable, length 29
IP 69.21.118.32 > 64.39.99.30: ICMP 10.4.191.24 protocol 250 unreachable, length 29
IP 69.21.118.55 > 64.39.99.30: ICMP 10.4.193.159 protocol 250 unreachable, length 29
IP 69.21.118.32 > 64.39.99.30: ICMP 10.4.191.24 protocol 251 unreachable, length 29
IP 69.21.118.55 > 64.39.99.30: ICMP 10.4.193.159 protocol 251 unreachable, length 29
IP 69.21.118.55 > 64.39.99.30: ICMP 10.4.193.159 protocol 252 unreachable, length 29
IP 69.21.118.32 > 64.39.99.30: ICMP 10.4.191.24 protocol 252 unreachable, length 29
```



In addition to providing valuable data to the sending host, ICMP error messages can be used by an attacker for reconnaissance activities. Port and protocol scans may lead to valuable information about listening services on the scanned host or network.

ICMP error messages may also provide more subtle information like even that the system is alive and responds.

ICMP Destination Unreachable Odd

Why do tcpdump and tshark get different results?

```
$ tcpdump -n -r icmp-error.pcap 'net 10' | wc -l
0

$ tshark -n -r icmp-error.pcap -Y 'ip.addr == 10.0.0.0/8' | wc -l
9
```

Do you see any 10.0.0.0/8 addresses?

```
$ tshark -n -r icmp-error.pcap -Y 'ip.addr == 10.0.0.0/8'
 2  0.125120 69.21.118.75 → 64.39.99.30  ICMP Destination unreachable (Protocol unreachable)
 3  6.125318 69.21.118.75 → 64.39.99.30  ICMP Destination unreachable (Protocol unreachable)
 4 12.125601 69.21.118.75 → 64.39.99.30  ICMP Destination unreachable (Protocol unreachable)
 5 19.031100 69.21.118.32 → 64.39.99.30  ICMP Destination unreachable (Protocol unreachable)
 6 22.346220 69.21.118.55 → 64.39.99.30  ICMP Destination unreachable (Protocol unreachable)
 7 27.030543 69.21.118.32 → 64.39.99.30  ICMP Destination unreachable (Protocol unreachable)
 8 28.347428 69.21.118.55 → 64.39.99.30  ICMP Destination unreachable (Protocol unreachable)
 9 34.347844 69.21.118.55 → 64.39.99.30  ICMP Destination unreachable (Protocol unreachable)
10 35.031022 69.21.118.32 → 64.39.99.30  ICMP Destination unreachable (Protocol unreachable)
```



There may also be unintended information leaks in ICMP error messages.

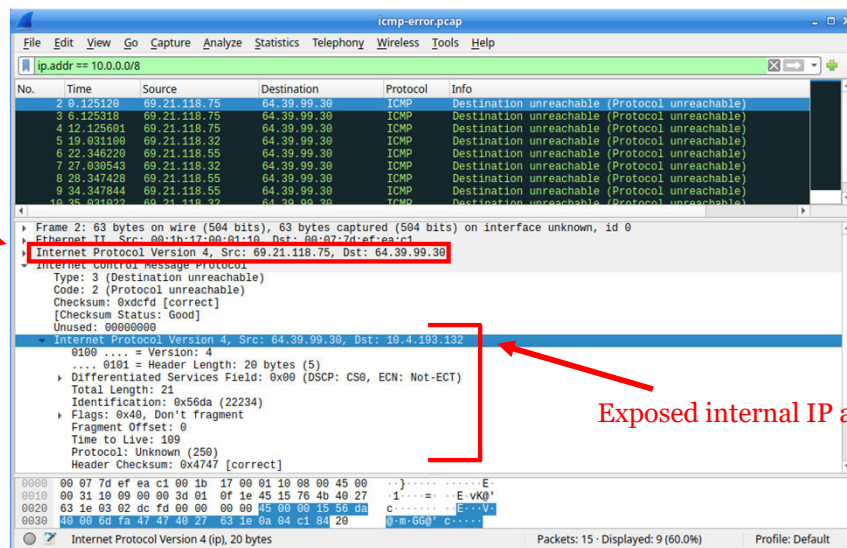
On the slide, tcpdump and tshark, a command-line version of Wireshark, are run against the icmp-error.pcap file. Each command is using a BPF or display filter to look for 10.0.0.0/8 network traffic.

It appears that tshark finds 9 packets with an IP address in the 10.0.0.0/8 network, and tcpdump did not find any. Looking at the bottom section of the slide, it appears that tcpdump has the correct answer.

So, why is tshark showing those 9 packets when there are clearly no 10.0.0.0/8 addresses?

ICMP Destination Unreachable Solved

IP header



Exposed internal IP address

SANS



icmp-error.pcap

SEC503: Intrusion Detection In-Depth 147

Wireshark and consequently tshark include the headers from ICMP error message payloads in display filter searches. This may be useful depending on the search criteria. However, it may also negatively impact the results if the search was for specific counts related to the actual header information.

This slide also highlights another potential risk. If a system is sitting behind a NAT (network address translation) device, is the NAT device aware of the header information contained in ICMP error messages? If aware, does the device properly update the ICMP error message hiding the internal IP addresses?

Mapping

Hosts

- ICMP Echo / Echo Reply
 - Single address, Broadcast address, Subnets
- Destination Unreachable
 - Host unreachable (inversely map)
 - Protocol unreachable
 - Port unreachable
- Reassembly time exceeded

Routers

- Destination Unreachable
 - Fragmentation needed, but DF bit was set
- Time Exceeded

Reconnaissance is an important part of a planned attack and usually performed in the early stages. Mapping may be used during reconnaissance to discovery systems and services in the network.

ICMP echo and echo replies are one of the most common mapping techniques, but there are other possible ways to use ICMP for reconnaissance. Unreachable messages can be used to find systems and services that are not available and conversely may lead to the ones that are available.

In addition to finding live hosts, ICMP error messages could be used to specifically attempt to find routers and potentially map a network. The time exceeded in transit messages are sent by routers, like in the tracert example. The fragmentation required but DF bit set allows discovery of the MTU of a link in the network. Routers would reply with these errors.

Conditions When ICMP Should Not Be Sent

- When error condition is temporary
- An ICMP error message should not be sent in response to:
 - Another ICMP error message
 - A destination broadcast address
 - A source address of broadcast or loopback address
 - Any fragment but the zero offset
- Why is it nonsensical to send an ICMP error back for a bad checksum?

Okay, now that we have a handle on ICMP messages and when they are sent, there are conditions when ICMP messages should *not* be sent. This gives you a more comprehensive view of ICMP. We mentioned that ICMP messages should be sent for non-transient conditions only. In other words, if the condition is temporary and likely to be corrected without an error message, ICMP messages are not sent.

ICMP messages should not be sent in response to receipt of another ICMP error message. For instance, say someone spoofs a packet from an existing source IP address and sends it to a non-listening destination port. The destination host reports that the port is unreachable and sends it back to the alleged sender. If the alleged sender tries to respond that it did not send the packet, it is possible that the two hosts could endlessly bounce errors back and forth.

Another condition when an ICMP error message should not be sent is in response to an IP packet with a destination broadcast address. As an example, suppose a UDP packet is sent to a broadcast address. Any host that does not listen on the destination port would return an ICMP destination port unreachable error, generating unnecessary traffic and potentially flooding the local network. Also, no ICMP error messages should be generated in response to a packet with a source address of the broadcast or loopback address. These are most likely maliciously generated packets in most cases, and there is a possibility of a denial of service if ICMP error messages are sent to these addresses.

Finally, only the initial or 0-offset fragment in a fragment train should generate an ICMP error message. You might wonder why this is so. That is because it is the only one that contains the protocol header; all others contain the data only. The host that receives the ICMP fragment error uses this embedded protocol header to associate it with the packets(s) that generated the error.

Test your overall comprehension of TCP/IP for a moment. Why would it make no sense to send an ICMP error in reaction to a bad checksum? If you aren't immediately sure, think about what it could mean if the checksum is invalid.

RFC 1122 states conditions when ICMP messages should not be sent.

ICMPv6

- Substantial ICMPv6 changes
- Added functionality:
 - Neighbor Discovery Protocol (NDP)
 - Multicast group membership (IGMP)
- Many ICMP type values changed:
 - 0-127: Error messages
 - 128-255: Informational

Look carefully at those number ranges; can you tell how the ranges are determined?

TCP and UDP are no different in IPv6 than they are in IPv4. In other words, no new or altered functionality is found in these protocols (except for jumbogram specification), as well as most other protocols that ride over IP. However, ICMPv6 has changed significantly. Let's just review some of the changes.

First, Neighbor Discovery Protocol (NDP) provides hosts on a local network a function similar to ARP in finding an address of a host or node on the same link. But unlike ARP, it also provides a means to determine which nodes are reachable, and it also has the capability to detect changes in link layer addresses. NDP can also find duplicate IP addresses in use on the same link. NDP includes a feature for hosts to solicit addresses of routers on the local network and for routers to advertise their presence.

In IPv4, multicast *Internet Group Management Protocol (IGMP)* was assigned its own protocol value of 2. In IPv6, multicast functions known as *Multicast Listener Discovery (MLD)* are incorporated into ICMP. This feature is used to determine multicast listeners on a link or whether there is a specific address on a link. Multicast addresses and functionality are used more routinely in IPv6 than IPv4.

You'll also find that the standard IPv4 ICMP types have changed. The ICMPv4 type 8 is an echo request, and type 0 is an echo reply. Now ICMPv6 type 128 is an echo request, and type 129 is an echo reply. Error messages have type values less than 128 and should not be blocked. Informational messages are associated with type values greater than or equal to 128 and may be more secure to block them inbound and outbound.

If you look at the number ranges carefully, you'll realize that there was some careful thought put into selecting them, unlike what we find with ICMP in IP version 4. The high-order bit is used to determine whether you are looking at an error (0) or an informational message (1).

UDP-ICMP Exercises

Workbook exercises "UDP-ICMP"

This page intentionally left blank.

Review: Fundamentals of Traffic Analysis: Part II

- Many different Wireshark display filters are available and accessible in different ways.
- Tcpdump filters are useful for extracting packets based on header values.
- TCP is a reliable protocol; UDP is not. And ICMP is used to convey error conditions and informational messages in IPv4 and IPv6 and used for Neighbor Discovery Protocol (NDP) in ICMPv6.

Let's wrap up with a summary of what was covered in this section. We began with using Wireshark display filters to pick out records with some distinguishing characteristics. Wireshark offers several different ways to compose display filters.

Next, we moved on to everyone's hands-down favorite section on tcpdump filters. Designating many of the filters is straightforward; however, they become complex when you want to examine a field that is less than a byte in length. A mask byte may be necessary to isolate the bits of interest.

We looked at embedded protocols next—namely, TCP, UDP, and ICMP. TCP is the most widely used transport protocol because it has mechanisms to ensure reliability. These come with the price of added complexity and less efficiency. UDP is a simple transport protocol that is used when an exchange can tolerate packet loss.

Finally, we explored ICMP in more detail to discover that it is used to report about non-transient problems as well as carry informational messages in both IPv4 and IPv6. ICMP has an expanded role in ICMPv6, as it carries NDP traffic.

Section 14.2 Packet Protocols as a Second Language

• An immersion class in speaking packet 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 Protocols as a Second Language

- Wireshark Display Filters
- Writing tcpdump Filters
- TCP
- UDP
- ICMP
- IPv6

Fundamentals of Traffic Analysis: Part II

IPv6 Fundamentals

For years and years, we've heard that IPv6 is right around the corner and its implementation is imminent. It's taken far longer than expected just because of all the infrastructure and implementation issues. But it is becoming more imperative now that the IPv4 addresses are gone. Some of the more agile or future-looking sites have already deployed IPv6. This section introduces you to the basic IPv6 concepts.

This section also discusses the obvious need for IPv6, looks at some of the major differences between it and its predecessor IPv4, and examines the new IPv6 address format, Neighbor Discovery Protocol, to find other hosts and routers on the local network, the structure of IPv6 packets, and the concept of extension headers to chain protocols and transition issues surrounding the deployment of IPv6.

You are most likely mistaken if you believe you have no IPv6 traffic on your network. Modern operating systems support it and generate traffic. You or your IDS/IPS should be examining it because your enemies definitely will try to find it and use its weaknesses for attack or use its invisibility in your network to exfiltrate.

Why IPv6?

Commonly cited, but wrong:

- Address exhaustion—we really aren't feeling this pressure yet.
- Security—security features are optional, not required.

Much more efficient

- Simpler and faster routing
- Eliminates broadcasts from all subnets

Better quality of service features

Larger packet payloads

IPv4 was not created with the notion that the internet would grow to be an international sensation, complete with stationary computers, mobile devices, thieves, and miscreants. And although it has provided adequate service for several decades, there are inherent problems that must be fixed.

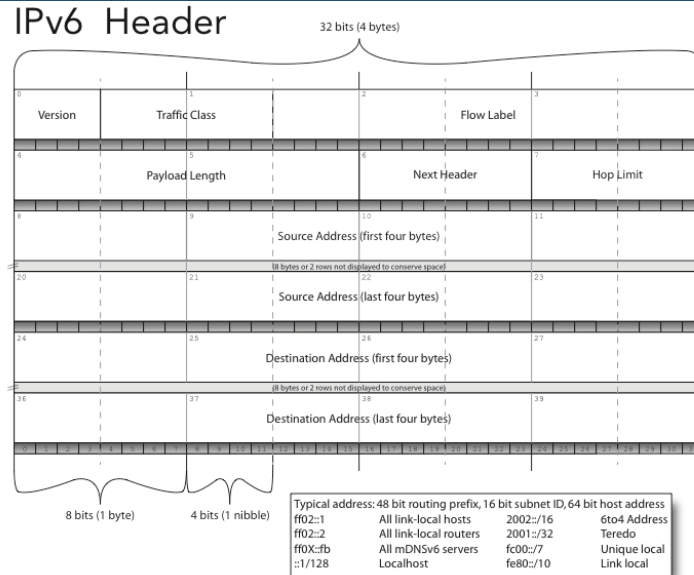
Some of the reasons that you have heard about that you might believe are driving IPv6 adoption are the IPv4 address space exhaustion problem and that IPv6 is much more secure. Both of these are incorrect. While the 32-bit addressing scheme no longer provides enough combinations of IP addresses to accommodate the growth of the internet and all of the many devices that now need addresses, you can request and obtain an address allocation of almost any size from an address broker within minutes to hours or days depending on the size of the allocation you are seeking. As for security, it is true that IPv6 *provides* for more security options, but none of these are *required*. Perhaps more eye-opening is the fact that a number of the network layer attacks that were resolved in IPv4 decades ago have been rediscovered within IPv6. It is our prediction that many more issues are lurking in IPv6 today, but we will not truly see them identified and resolved until IPv6 is adopted *much* more widely.

The real benefits of IPv6, and the reason that you should be trying to migrate to it now, are performance related. IPv6 is much more efficient than IPv4. Consider:

Every router is encumbered by the requirement to recompute the IPv4 checksum because it decrements the TTL. Routers are also burdened with having to fragment packets that are larger than the support MTU. Finally, the size of the routing tables in IPv4 is unwieldy. IPv6 advocates route aggregation (creating simple hierarchical routes) so that the number of routes that need to be stored in the routing table is more manageable.

Better quality of service features needed for real-time applications are also available. And now with jumbo packets, the 65535-byte limitation on the packet size has been increased in IPv6. These are just a few of the many reasons for the creation of IPv6.

IPv6 Header



The transition to IPv6 *will* happen; it's just a matter of how rapidly and how prepared you and your organization will be. For reference, we have included a typical rendering of the IPv6 header.

Please note that, to conserve space, the source and destination address lines have been compressed!

IPv4 Versus IPv6

	IPv4	IPv6
IP Header Length	Greater than or Equal to 20 Bytes	40 Bytes
IP Address Length	32 Bits	128 Bits
Length	Packet Size	Payload Size
Name Changes		
	TTL	Hop
	Type of Service	Traffic Class
	Protocol	Next Header

Probably the most-discussed feature of IPv6 is the extended address space (more later). However, a few other things have been "fixed." The header is now limited to exactly 40 bytes. Instead, additional "extension headers" are used to express functionality covered by IPv4 fields and options. This format is intended to ease parsing of the header and transmission speed. Routers will have to consider the 40-byte IPv6 header only to pass a packet on.

What used to be *Type of Service* is now known as *Traffic Class*, allowing for 256 different traffic classes. These traffic classes can adjust *quality of service (QoS)* parameters in routers. Although similar in function to the IPv4 TOS field, it provides for many more variations.

The IPv4 protocol field that indicates the protocol that follows the IP header is now known as the *Next Header* field in the IPv6 header. In addition to indicating the transport layer protocol, it may also indicate the type of *extension header*, if there is one. Some of the same protocol numbers from IPv4 are still used, but some change for IPv6, notably ICMPv6.

The IP packet header length has been deleted from the IPv6 header because the header is a fixed 40 bytes. If any of the old IP options are required, they are implemented in an extension header that follows the IPv6 header and precedes the transport layer of the packet. Other fields (fragmentation offset, IP ID, and flags) all support fragmentation, another extension header. A packet can have multiple extension headers. Finally, the IPv4 checksum that validated the IPv4 header is gone. Now, the protocol checksums (TCP and UDP and ICMP, for instance) are considered sufficient because they have pseudo-headers that validate portions of the IPv6 header. This also relieves routers from having to recompute IP checksums after decrementing the TTL (now known as the HOP limit value).

The packet length field underwent a subtler change. In IPv4, this field indicates the length of the entire packet, including the IPv4 header. In IPv6, the field was renamed to *payload length* and does not include the IPv6 header size.

Fields removed: Fragmentation, Flags, Checksum, Options, IP ID, Length

Fields added: Flow Label

IPv6 Addresses Components

Prefix Subnet ID Interface ID

AAAA:AAAA:AAAA:BBBB:CCCC:CCCC:CCCC:CCCC

Prefix

- The routing prefix; a company network may be assigned one or more prefixes.
- The first byte indicates the type of address (link local, globally routable, etc.).

Subnet ID

- Identifies a particular subnet; your minimum allocation is 65535 subnets.

Interface ID

- Every host is assigned at least one address; may be derived from the MAC (EUI-64).
- Privacy Enhanced mode can randomize the lower 64 bits of the address.

It is up to the registrar (ARIN, APNIC, RIPE, etc.) to decide how to divide the address space. ARIN currently allocates IPv6 blocks with a minimum initial size of /32. Additional blocks are available with a size of /48. ARIN further specifies that most "sites" (for example, a company) should be allocated a /48. A /64 is to be used if only one subnet is needed (for example, for a consumer; see <http://www.arin.net/policy/nrpm.html#six>).

The interface ID may be derived from the MAC address using a scheme known as EUI-64 format:

- Split the MAC into two 6-byte pieces.
- Insert the bytes 0xFFFE between the two 6-byte pieces.
- Invert the 7th bit (from the left) in the resulting interface ID; this is also the 7th bit in the original MAC.

Why is this bit flipped? Some of the bits in the MAC have special legacy meanings. This bit is used to indicate whether the address is a local address or a universal address. Local addresses have this bit set; universal addresses have this bit cleared. In practice, what this means is that the MAC address 00:50:56:C0:00:08 would have ::0250:56FF:FEC0:0008 as the lower 64 bits in an IPv6 address.

One of the concerns with the stateless autoconfiguration of IPv6 addresses is that the host's MAC address can be derived from the lower 64-bit EUI value. As you know, in IPv4, MAC addresses can be paired with an IPv4 address within a local network only. This address is never seen outside of the local subnet.

An outbound IPv6 address that isn't NAT'd could identify a specific physical host. This raises user privacy concerns. This issue has been resolved via a privacy extension to the autoconfiguration algorithm to randomize the lower 64 bits of the address. These addresses can be configured to expire to provide the user more privacy.

Address Types

	IPv4	IPv6
Loopback	127/8	::1/128
Link Local	169.254/16	fe80::/10
Global Unicast	Any Routable IP Address	2000::/3
IPv4-Mapped	N/A	::ffff/96
IPv6 to IPv4	N/A	2002::/16
Documentation	192.0.2/24, 198.51.100/24, 203.0.113/24, 233.252.0/24	2001:0db8::/32
Multicast	224.0.0.0/4	ff00::/8
Private	10/8 192.168/16 172.16/12	fc00::/7
Many reserved ranges	0/8, 100.64/10, 192.0.0/24, 192.18/15, 240/4	2001::/32, 2001:20::/28, 4000::/3, 6000::/3, 8000::/3, a000::/3, c000::/3, e000::/4, f000::/5, f800::/6, fe00::/9, fec0::/10

There are several different address types for both IPv4 and IPv6. IPv4 knows *unicast*, *multicast*, and *broadcast* addresses. IPv6 no longer uses broadcast. For the larger flat networks proposed as part of IPv6, broadcast in its original use would be inappropriate. The closest thing to broadcast is *link local*. Link local is limited to a physical link, not necessarily to a logical network address block.

A *global unicast* address is one that can be connected to the internet and is routable. IPv4-mapped addresses are used to embed an IPv4 address in an IPv6 address. This can be used when transitioning from a dual-stack host. IPv6 to IPv4 is a method for IPv6 to be carried over IPv4, typically when IPv6 is supported between the sending and receiving networks, but not the route between.

The *documentation/example* category is used when you want to refer to a legitimate example network. For instance, in this course, any fictitious traffic that is created for the purpose of documenting should use these ranges (though we didn't follow this suggestion... nor do most people). You are probably familiar with multicast to send traffic to a group of multicast hosts. And the private networks are ones used locally and not intended to be routed over the internet.

You might notice that there are many reserved ranges both in IPv4 and in IPv6. This might surprise you in IPv4, where there are millions of unused "reserved for future use" addresses that, apparently, will never be used. IPv6 also seems to have huge ranges of reserved addresses. In the case of IPv6, this is for planned future expansion. Currently, all our publicly routable addresses are in the 2000::/3 range (so, that would be prefixes beginning 2xxx and 3xxx). The many /3 ranges that are reserved are expected to be used to allocate additional globally routable addresses when they become needed. Given the number of addresses, that is likely going to be when we get to IP Galactic Version™. ☺

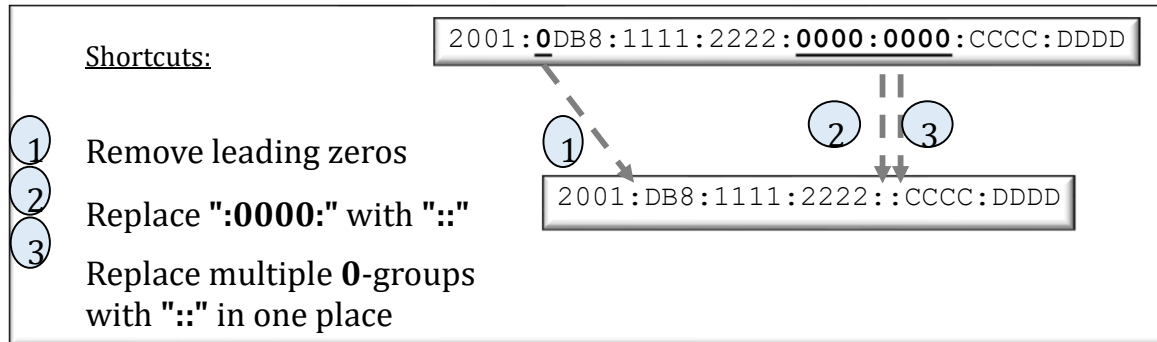
IPv6 Unicast Addresses

IPv6 address 128 bits =

340,282,366,920,938,463,463,374,607,431,768,211,456 available

IPv4 address 32 bits = 4,294,967,296 available

IPv6 cumbersome notation, shortcuts can be used



IPv6 addresses are 128 bits versus 32 bits for IPv4 addresses. There are now 340,282,366,920,938,463,374,607,431,768,211,456 combinations available, while there were 4,294,967,296 available for IPv4.

Remembering and expressing IPv6 addresses is challenging. Typically, they are written in eight groups of 2 bytes of hexadecimal. A couple of shortcuts can be used. For example, leading zeros may be dropped. Groups of zeros may be replaced with **"::"**. However, this can be done only once per IP address to avoid ambiguity. Consider this example:

2001:0DB8::2222:3333::4444 ← DOES NOT WORK

because it could either stand for:

2001:0DB8:0000:0000:2222:3333:0000:4444 or

2001:0DB8:0000:2222:3333:0000:0000:4444

Subnets are noted like in IPv4, but the trailing zeros may be dropped (that is, 2001:0DB8/32).

The vastness of the IPv6 space has been described with many different analogies, being compared to the number of stars in a galaxy or grains of sand on the planet.

What Shouldn't We See?

Unnatural values for source IP numbers entering network:

- IP numbers that fall in your network range
- Any source that isn't globally routable
- Your globally routable addresses

Unnatural values for destination IP numbers entering network:

- Any address that isn't globally routable
- Any globally routable address that isn't you

Unnatural values for source IP numbers leaving network:

- IP numbers that don't fall in your network range
- Source IP that isn't globally routable

Unnatural values for destination IP numbers leaving network:

- Any address that isn't globally routable

The source IP number is located in the 8th–23rd bytes offset of the IPv6 header; the destination IP number is located in the 24th–39th bytes offset of the IPv6 header. The same caveats apply to inbound and outbound IP addresses found in IPv6 as IPv4. Make sure traffic leaving your network has a legitimate source and destination IPv6 address; otherwise, block it. The same applies to inbound traffic as well.

For more information on IPv6 address assignments:
<http://www.iana.org/assignments/ipv6-address-space>

IPv6 Checksums

No IP checksum in IPv6 header:

- Smaller IPv6 header
- More efficient!
 - Only calculated at endpoints
 - Encapsulated protocols responsible for validation

Embedded protocols must implement a pseudo-header checksum.

- This means that the checksum must include portions of the IP6 header.

There is no longer an IP checksum associated with IPv6 traffic. This eliminates a field in the IPv6 header, making it more compact. Also, there are big efficiency gains in terms of routing. If you recall, each router after decrementing the TTL had to recompute the IPv4 header checksum because a field in the IP header had changed. Routers that support IPv6 traffic no longer have to do this.

What happens if there is corruption in a field in the IPv6 header? If the receiving protocol wants to ensure that the IP header has not been altered in transit, it must support some kind of checksum computation. We have not discussed pseudo-header checksums yet because they apply to embedded protocols following the IP header that we will discuss later in the course. However, the concept is that these protocols include some vital fields from the IP header (IP addresses, protocol/next header) in their checksum computations. Although the IPv4 header checksum included all the fields in the IP header, those computed by the IPv6 pseudo-header checksum are the ones that are most vital for getting the traffic from source to destination uncorrupted. Essentially, the function of validation of the vital IPv6 header values has shifted from the IP layer to the transport layer.

IPv6 and Common Protocols (I)

No substantial changes to TCP and UDP:

- Same next header (NH) value (equivalent of IP header protocol value) as IPv4:
 - NH of 6 = TCP
 - NH of 17 = UDP
- Retrofits to accommodate jumbograms:
 - TCP maximum segment size of 65535
 - UDP header length of 0

TCP and UDP remain mostly the same in IPv6 as in IPv4. In IPv6, there is a concept of the next header, or NH, to identify the protocol that follows the current header. IPv4 had a similar concept known as the protocol field in the IPv4 header, which held the value of the header type that followed the IPv4 header. The common protocol values in IPv4 are 1 for ICMP, 6 for TCP, and 17 for UDP. The TCP and UDP values have not changed. The protocol/next header value for ICMPv6 is 58.

There have been some accommodations for the IPv6 jumbogram—a packet that is larger than 65535 bytes. There are retrofits to the established protocols to designate the presence of a jumbogram. IPv6 TCP indicates that a jumbogram follows with a TCP option maximum segment size (MSS) value of 65535. Jumbograms should be used only when the link MTU can support them.

UDP makes use of the length value in the UDP header to signal that the packet is a jumbogram. The UDP header length reflects the size of the UDP header (standard 8 bytes) plus the size UDP payload follows. The minimum value is 8 because a UDP header is required. A value of 0 in the UDP length field means that the packet is a jumbogram.

IPv6 and Common Protocols (2)

Substantial ICMPv6 changes:

- New NH value = 58, IPv4 protocol value = 1
- ICMPv6 types changed:
 - Unreachable = type 1 (IPv4 = 3)
 - Echo Request = type 128 (IPv4 = 8)
 - Echo Reply = type 129 (IPv4 = 0)
 - Neighbor Solicitation = type 135: Request link layer address (like ARP request)
 - Neighbor Advertisement = type 136: Response (like ARP reply)

IPv6 significantly extends the use of ICMP, also known as ICMPv6. ICMPv6 is identified in the NH field in the IPv6 header with a value of 58. Remember, IPv4 identified ICMP with a protocol field in the IP header with a value of 1.

All functions available in ICMP version 4 are still available in ICMPv6, but new "types" have been added in the reserved range of values from 42–255 to facilitate the new functions of IPv6:

128: Echo Request

129: Echo Reply

133: Router Solicitation (for autoconfiguration)

134: Router Advertisement (for autoconfiguration)

135: Neighbor Solicitation (used to be done via ARP)

136: Neighbor Advertisement (used to be done via ARP)

Types 1–127 represent ICMPv6 error messages and should not be blocked. Types 128–255 are informational messages and may be considered for blocking.

ARP Goes Away: Neighbor Discovery Protocol

Host-to-host

- Address resolution
- Next-hop discovery
- Neighbor Unreachability Detection (NUD)
- Duplicate Address Detection (DAD)

IPv6 uses ICMPv6 *Neighbor Discovery Protocol (NDP)* to perform many of the same functions done in IPv4 using *Address Resolution Protocol (ARP)* and ICMP router discovery and redirect. However, NDP treats these functions, as well as others, as an integral part of discovery and not as an afterthought like IPv4. NDP is not actually a protocol per se, but it is a set of functions to perform some routine network tasks.

Address resolution to associate an IP and its related MAC address in IPv6 involves two different ICMPv6 exchanges: a *Neighbor Solicitation (NS, ICMPv6 type 135)* sent to the **all hosts** multicast address and a *Neighbor Advertisement (NA, type 136)* from the target host to return its link layer address. This is similar in concept to ARP but is performed using ICMPv6.

Next-Hop determination is performed by comparing the prefix information received from the local router with the destination address. If they differ, the next hop is selected from a manually configured list of local routers or via the host-router *Neighbor Discovery*. If they do not differ, the destination host resides on the same local network. The gathered next-hop data is cached for efficiency.

Neighbor Unreachability Detection provides the capability of determining if a neighbor is no longer reachable. When possible, this is done using upper-layer protocol confirmation to a host, such as a TCP acknowledgment by the host. If this is not available, a unicast Neighbor Solicitation is sent to neighbors to check each neighbor cache for a recent entry reflecting communication with the target host.

Duplicate Address Detection (DAD) ensures that two devices do not assign the same IP address. This is performed when a new host joins the network and sends a Neighbor Solicitation for its IP address. If it receives a Neighbor Advertisement from another host, it knows that the IP address is already in use.

If you'd like more details on NDP, look at RFC 2461, "Neighbor Discovery."

Where's My Default Gateway?

Host-to-router

- Router discovery
- Prefix discovery
- Parameter discovery
- Address autoconfiguration
- Redirect

Let's look at NDP communications between a host and a local router. IPv6 uses a *Router Solicitation* (RS, type 133) and *Router Advertisement* (RA, type 134), much like the Neighbor Solicitation/Advertisement NDP protocol. When a host first becomes part of a link, a multicast Router Solicitation is sent to the routers. Any listening router replies with a Router Advertisement sent to all hosts. This message contains the MAC address of the router, one or more prefixes/subnets that represent the local network that do not need to be routed (*Prefix Discovery*), and the MTU of the link to the router, along with an initial hop count value (same as IPv4 TTL) for the host to use (*Parameter Discovery*).

Address Autoconfiguration provides *stateless* address configuration. It is called stateless because it requires no other information or interaction with any other node on the network. This may be a temporary address assigned if *stateful* configuration is later assigned by a DHCPv6 request. In between the stateless and stateful assignments, the host uses DAD to make sure there are no duplicates in the network and make a Router Solicitation to elicit information about its network.

Address Autoconfiguration generates a link local address. The first 10 bits of the address are 1111 1110 10, followed by 54 zeros and the lower 64 bits derived from the MAC address using EUI-64 formatting.

The NDP *Redirect* function is similar to the IPv4 redirect, where a router informs a host of a better route. We'll discuss the concept of router redirect when we cover ICMP.

Multicast is Heavily Leveraged

All IP6 addresses beginning FF are multicast

- | | |
|-------------|---------------------|
| • ff00::/12 | Reserved |
| • ff01::/12 | Interface-local |
| • ff10::/12 | Transient addresses |
| • ff02::/12 | Link-local |
| • ff05::/12 | Site-local |

Good to know:

- | | |
|-------------|----------------------------|
| • ff02::1 | All nodes |
| • ff02::2 | All routers |
| • ff02::4-A | Specific Routing Protocols |
| • ff02::FB | Multicast DNSv6 |

While the address space is vast, much of it is highly structured. For example, any address beginning with the byte 0xFF in the prefix is a multicast packet of some kind. Any packet beginning with 0xFE is some type of link-local destination. The multicast addresses are of particular interest for us because multicast is very heavily leveraged in IPv6. This is especially true for the various ICMP discovery protocols that are now used for discovering hosts, routers, name servers, and more.

The structure doesn't end there. Within the multicast allocation, the next byte defines the scope of the rest of the address. For our purposes, link-local and site-local are the most important. What we state about the link-local addresses can be directly applied to the site-local addresses (and all other scopes, for that matter) as well.

Within the local network (subnet, or LAN), we can easily address all hosts, routers, types of routers, SSDP hosts, uPNP hosts, DHCP-related functions, DNS servers, and more. This structure formalizes what many network administrators have tried to do over the history of IPv4—that is, create conventions for the addressing and locating of hosts within an enterprise environment. IPv6 formalizes this into the addressing!

The Solicited-Node Multicast address, ff02::1:ff/24, can be used to find any host on the local network whose interface address ends with the last 3 bytes of this address. This turns out to be extremely powerful.

Solicited-Node Multicast

The address ff02::1 represents the multicast address for all hosts.

- Multicast address of ff02::1:ff/104 is the solicited-node multicast address

Node's unicast and anycast addresses required to have an associated solicited-node multicast address to join the group

Solicited-node multicast address derived by:

- Taking low-order 24 bits of unicast/anycast address
- Appending them to the prefix of ff02::1:ff00:0

Uncompressed

```
ff02:0000:0000:0000:0000:0001:ff00:0000
fe80:0000:0000:0000:784b:4d2c:14a7:9ad5
ff02:0000:0000:0000:0000:0001:ffa7:9ad5
```

Compressed

```
ff02::1:ff00:0
fe80::784b:4d2c:14a7:9ad5
ff02::1:ffa7:9ad5
```

Consider the address fe80::784b:4d2c:14a7:9ad5. The solicited-node multicast address for this host would be ff02::1:ffa7:9ad5. Notice that the final 24 bits from the unicast address have been copied into the solicited-node address. If you recall, the ff02::1 address represents the multicast address to all hosts. A multicast address of ff02::1:ff is the solicited-node multicast address. It is used specifically for NDP. A node's unicast and anycast addresses are required to have an associated solicited-node multicast address.

Why is there a need for a solicited-node multicast address anyway? IPv4 ARP uses a broadcast address as the destination IP address when trying to find a destination MAC address. This is sent to all hosts on the local network. Broadcast addresses do not exist in IPv6. Suppose they did and there was a local /64 subnet size. The number of listening hosts could potentially be 2^{64} , which becomes untenable in practice. Multicast addresses provide a manageable subset of hosts in a particular group, and this scheme ensures that the NS is sent to all hosts on the local network.

The process for deriving the solicited-node multicast first extracts the 24 low-order bits of the destination IPv6 address. We find a7:9ad5 in the 24 low-order bits of the desired destination address. The upper 104 bits are ff02:1:ff00:0 in compressed notation. The uncompressed 128-bit representations are shown in the left box at the bottom of the slide, with the intention of making the notation and process more obvious. The associated compressed notation is shown in the right box at the bottom of the slide. This same process must occur for each interface with a unicast or anycast address.

Neighbor Solicitation

No.	Time	Source	Destination	Protocol	Src Port	Dest Port	Info
1	2012-08-16 09:18:11.126672	fe80::224:8cff:fe2b:d64e	ff02::1:ffa7:9ad5	ICMPv6			Neighbor Solicitation
2	2012-08-16 09:18:11.129193	fe80::784b:4d2c:14a7:9ad5	fe80::224:8cff:fe2b:d64e	ICMPv6			Neighbor Advertisement
3	2012-08-16 09:18:11.129235	fe80::224:8cff:fe2b:d64e	fe80::784b:4d2c:14a7:9ad5	ICMPv6			Echo (ping) request id
4	2012-08-16 09:18:11.131864	fe80::784b:4d2c:14a7:9ad5	fe80::224:8cff:fe2b:d64e	ICMPv6			Echo (ping) reply id=6

▶ Frame 1: 86 bytes on wire (688 bits), 86 bytes captured (688 bits)
 ▶ Ethernet II, Src: aa:00:04:00:0a:04 (aa:00:04:00:0a:04), Dst: 33:33:ff:a7:9a:d5 (33:33:ff:a7:9a:d5)
 ▶ Internet Protocol Version 6, Src: fe80::224:8cff:fe2b:d64e (fe80::224:8cff:fe2b:d64e), Dst: ff02::1:ffa7:9ad5 (ff02::1:ffa7:9ad5)
 ▶ Internet Control Message Protocol v6
 Type: Neighbor Solicitation (135)
 Code: 0
 Checksum: 0x5089 [correct]
 Reserved: 00000000
 Target Address: fe80::784b:4d2c:14a7:9ad5 (fe80::784b:4d2c:14a7:9ad5)
 ▼ ICMPv6 Option (Source link-layer address : aa:00:04:00:0a:04)
 Type: Source link-layer address (1)
 Length: 1 (8 bytes)
 Link-layer address: aa:00:04:00:0a:04 (aa:00:04:00:0a:04)

Notice that the magic of multicast happens at layer 2! 33:33 is the multicast prefix.



Suppose IPv6 host fe80::224:8cff:fe2b:d64e needs to ping fe80::784b:4d2c:14a7:9ad5 yet has no entry in its neighbor cache where it stores IPv6 address and link address pairings. It must send a Neighbor Solicitation (NS) using an ICMPv6 message with type 135 to discover the destination host's MAC address. Take note of the target IPv6 address and compare it to the destination IPv6 address in the IPv6 header. Do you recognize the destination address? That is the solicited node multicast address for the target address!

It's also important to take note of the destination MAC. It turns out that the real magic of multicast is happening at layer 2. Switches are responsible for maintaining multicast channel membership and properly forwarding frames to hosts subscribed to those channels on the local network. The 33:33 prefix in a destination address indicates that this is a multicast address. The channel number is FF:A7:9A:D5, which is the end of the solicited node multicast address for the target host.

Think about what this does for us. Rather than sending a broadcast, which will consume a time slot on every link to every host on the subnet, a multicast frame is sent. This will be forwarded to every host on the subnet subscribed to channel FF:A7:9A:D5. How many hosts is that likely to be? Just one! This is a *huge* efficiency benefit over IPv4.

The source of the NS contains its MAC address, as you see in the ICMPv6 source link layer option. There are many ICMPv6 options that can be associated with NDP. We'll see some additional ones in the router advertisement ICMPv6 message.

To view the output, enter the following at the command line:

wireshark ip6-neighbor-solandadv.pcap

This is the first record.

Neighbor Advertisement

No.	Time	Source	Destination	Protocol	Source port	Destination port	Info
1	0.000000	fe80::224:8cff:fe2b:d	ff02::1:ffa7:9ad5	ICMPv6			Neighbor solicitation
2	0.002521	fe80::784b:4d2c:14a7:	fe80::224:8cff:fe2b:d	ICMPv6			Neighbor advertisement
3	0.002563	fe80::224:8cff:fe2b:d	fe80::784b:4d2c:14a7:	ICMPv6			Echo request
4	0.005192	fe80::784b:4d2c:14a7:	fe80::224:8cff:fe2b:d	ICMPv6			Echo reply

Frame 2 (86 bytes on wire, 86 bytes captured) Ethernet II, Src: IntelCor_43:9f:3e (00:21:6b:43:9f:3e), Dst: DigitalE_00:0a:04 (aa:00:04:00:0a:04) Internet Protocol Version 6 Internet Control Message Protocol v6 Type: 136 (Neighbor advertisement) Code: 0 Checksum: 0xc1f6 [correct] Flags: 0x60000000 Target: fe80::784b:4d2c:14a7:9ad5 (fe80::784b:4d2c:14a7:9ad5) ICMPv6 Option (Target link-layer address) Type: Target link-layer address (2) Length: 8 Link-layer address: 00:21:6b:43:9f:3e
--



If the target host is listening, it returns a Neighbor Advertisement message using ICMPv6 type 136. Also, for some added efficiency, all neighbors that see the NS message cache the source IP and MAC address pairing for the sender of the advertisement and maintain it in their neighbor cache for a limited time. The MAC address is returned in another ICMPv6 option: the target link layer address.

The two hosts can now communicate (in this case, with an ICMPv6 echo request and reply).

To view the output, enter this at the command line:

```
wireshark ip6-neighbor-solandadv.pcap
```

This is the second record.

Router Solicitation

No.	Time	Source	Destination	Protocol	Source port	Destination port	Info
1	0.000000	fe80::21b:63ff:fe94:b10e	ff02::2	ICMPv6			Router solicitation
2	0.001470	fe80::21b:90ff:fe2d:e	ff02::1	ICMPv6			Router advertisement

Frame 1 (70 bytes on wire, 70 bytes captured) Ethernet II, Src: AppleCom_94:b1:0e (00:1b:63:94:b1:0e), Dst: IPv6mcast_00:00:00:02 (33:33:00:00:00:02) Internet Protocol Version 6 Internet Control Message Protocol v6 Type: 133 (Router solicitation) Code: 0 Checksum: 0x51b2 [correct] ICMPv6 Option (Source link-layer address) Type: Source link-layer address (1) Length: 8 Link-layer address: 00:1b:63:94:b1:0e



Let's examine another exchange when a host joins a network. When host fe80::21b:63ff:fe94:b10e is first connected to the network or is newly rebooted, it must know its first-hop routers. It finds them by issuing a Router Solicitation message, ICMPv6 type 133, to the multicast address of ff02::2 that represents the "all routers" address. Again, you see the ICMPv6 option source link layer address with the sender's MAC address.

To view the output, enter this at the command line:

wireshark ip6-router-solandadv.pcap

This is the first record.

Router Advertisement

Time	Source	Destination	Protocol	Source port	Dest port	Info
0.000000	fe80::21b:63ff::ff02::2	ff02::1	ICMPv6			Router Solicitation from 00:1b:63:94:b1:0e
0.001470	fe80::21b:90ff::fe2d::e43	ff02::1	ICMPv6			Router Advertisement from 00:1b:90:2d:0e:43
▶ Frame 2: 118 bytes on wire (944 bits), 118 bytes captured (944 bits)						
▶ Ethernet II, Src: Cisco_2d:0e:43 (00:1b:90:2d:0e:43), Dst: IPv6mcast_00:00:00:01 (33:33:00:00:00:01)						
▶ Internet Protocol Version 6, Src: fe80::21b:90ff::fe2d:e43 (fe80::21b:90ff::fe2d:e43), Dst: ff02::1 (ff02::1)						
▼ Internet Control Message Protocol v6						
Type: Router Advertisement (134)						
Code: 0						
Checksum: 0x1b13 [correct]						
Cur hop limit: 64						
▶ Flags: 0x00						
Router lifetime (s): 1800						
Reachable time (ms): 0						
Retrans timer (ms): 0						
▶ ICMPv6 Option (Source link-layer address : 00:1b:90:2d:0e:43)						
▶ ICMPv6 Option (MTU : 1500)						
▼ ICMPv6 Option (Prefix information : 3ffe:80c0:22c:190::/64)						
Type: Prefix information (3)						
Length: 4 (32 bytes)						
Prefix Length: 64						
▶ Flag: 0xc0						
Valid Lifetime: 2592000						
Preferred Lifetime: 604800						
Reserved						
Prefix: 3ffe:80c0:22c:190:: (3ffe:80c0:22c:190::)						



The router with an IP address of fe80::21b:90ff::fe2d:e43 responds with a Router Advertisement, ICMPv6 type 134, to the destination address of ff02::1, the multicast address of all hosts. This is sent as a multicast message, not a unicast message, so that any listening hosts may learn this same information if they do not already know it.

The router responds with some information in addition to its address. The *Cur hop limit* of 64 informs hosts that that is the hop limit value to place in the IPv6 header. If you recall, this is equivalent to the IPv4 Time to Live value that is used to designate the maximum hops/routers that a packet can traverse before being expired.

The router lifetime is the number of seconds that the information in this advertisement should be kept. This particular router information is active for 1800 seconds, after which the host(s) must make another Router Solicitation. The time information that follows informs the receiving hosts how long to consider a neighbor reachable and the time, in milliseconds, that a host needs to wait before attempting to retransmit NS messages.

There are two other ICMPv6 NDP options: the MTU and Prefix options. The MTU specifies the maximum size a packet on the network must be to avoid fragmentation. The Prefix option designates the prefix to be used in the IPv6 global/routable address when stateless address autoconfiguration is used. If you recall, there are several different IPv6 type addresses for a given interface. The link-local ones begin with fe80:: and are used for local/nonroutable traffic. The prefix is applied to the global/routable IPv6 address for the interface. In this case, the prefix is the 64 high-order bits of 3ffe:80c0:22c:190::/64.

To view the output, enter this at the command line:

```
wireshark ip6-router-solandadv.pcap
```

This is the second record.

Neighbor Discovery Attacks

Neighbor Solicitation: Spoof NA with wrong link address

DoS: Respond to NS with nonexistent link address

Neighbor Unreachable Detection: Spoof response to NS with address not reachable

Duplicate Address Detection: Spoof NS response with address already taken

Router Solicitation: Spoof router advertisement to indicate attacker's IP address is the next-hop router

Much like the issues with IPv4 ARP, and ICMPv4 router messages, the presence of an attacker on an IPv6 network can disrupt and redirect legitimate traffic. This is easily done using NDP messages because there is no inherent security ensuring that the message is not spoofed by a malicious node.

An incorrect IPv6 host address/link layer pairing can be performed in one of two ways. The first and more obvious way is for the malicious host to respond to a Neighbor Solicitation (NS) request, spoofing a bogus address in the returned Neighbor Advertisement (NA). This was known as *ARP poisoning* in IPv4. A second way is to send an incorrect IPv6 host address/link layer pairing in the NS request itself because neighbors seeing this will place the pairing in their neighbor cache. This is similar to the use of a spoofed gratuitous ARP in IPv4. Another attack, a DoS, can be performed when the NS request receives a spoofed NA response indicating that the target host is at a nonexistent link address, denying communication with the target host.

An analogous attack is the use of the Neighbor Unreachable Detection to return a spoofed NA response that the target host is unreachable. If you recall, a new node joining a network sends an NS message to determine if other hosts have a duplicate address. A DoS can be attempted if the malicious host can spoof a response indicating that the address is already in use via the Duplicate Address Detection. If the originating host attempted to generate one or more new addresses, the malicious host could respond again with DAD, effectively denying the host an address and the capability to communicate on the network.

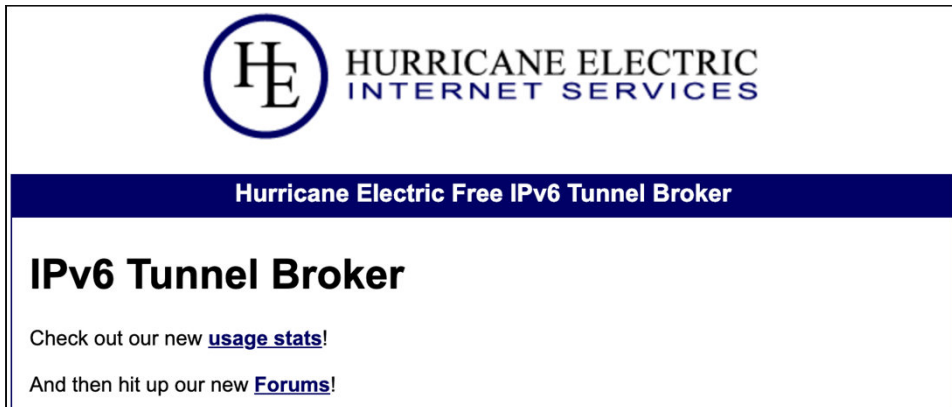
Finally, when a host joining a network makes a Router Solicitation to discover its router(s), a spoofed Router Advertisement could indicate that the attacker's host is the router, thereby routing all traffic through it, creating a MITM attack.

These are just a few of the many NDP attacks that can be performed. If you are interested in reading about others and proposed fixes, see RFC 3756, "IPv6 NDP Trust Models and Threats."

IPv6 and You

Your organization should be using IPv6 today.

- If you aren't, at the very least set up a test lab!
 - Firewall in a lab providing an IPv6 LAN connected to an IPv6 tunnel broker!



Most organizations are very much still in an IPv4 world... and we expect that this will remain true for some time. Yet, we also know that they will *definitely* migrate to IPv6 at some point, likely within the next five to ten years. Is your organization ready? Have you done any kind of testing? Think about applications written within your organization in the last five years. Have the programmers even thought about providing the ability to specify an IPv6 address?

One way to begin making steps forward, even if your ISP doesn't support IPv6, is to set up a test IPv6 network. This is simple enough internally, but it is difficult to really come to a good understanding of IPv6 if you can't talk to the rest of the world using it. To overcome the lack of a public IPv6 connection, you should consider creating an account with an IPv6 tunnel broker. Hurricane Electric is one such provider.

Hurricane will allow you to create an IPv6 tunnel at no cost. You will be assigned an entire IPv6 subnet over which you have complete administrative control. This can then be used to simulate what a real IPv6 internet connection would look like and allow you to experiment with your important services in a realistic test bed.

If you take this route, we recommend that you use an internal firewall as the endpoint for your tunnel. This allows you to insulate your network from what is effectively a publicly routed network within your IPv4 network.

IPv6 Review

Necessary to expand number of available IP addresses, yet benefits for routing efficiency, too

IP header changed:

- Standard 40 bytes
- Fields and functions moved to extension headers

ICMPv6 has many more functions than ICMPv4.

Transition from IPv4 to IPv6 can use tunnels in the interim before IPv6 support standard

The proliferation of IP-enabled devices has caused the exhaustion of IPv4 32-bit addresses. IPv6 supplies the expansion of the available IP addresses, accomplished with a 128-bit address. Routing schemes have been simplified as well for more efficiency.

One of the most significant changes with respect to IPv4 is that the IPv6 header is limited to exactly 40 bytes. Extension headers are used in place of IP options fields, and fragmentation has a separate extension header instead of being included in the standard header.

As the transition from IPv4 to IPv6 occurs, there are technologies to enable communications between the two. These include IP tunnels capable of carrying IPv6 packets over IPv4 packets, where one or both gateways communicate using IPv4 only and endpoints are IPv6 enabled. Teredo provides possible unknown and unsafe IPv6 host communication by tunneling IPv6 packets over IPv4 UDP.

Review of Fundamentals of Traffic Analysis: Part I

Network traffic layered

Encapsulation/de-encapsulation requires knowledge of layers that follow and lengths involved

ARP used to translate IP to link layer addresses

IP concerned about next-hop transit

Fragmentation necessary when packet length exceeds link MTU

Let's highlight some of the topics covered in Section 1. We examined the concept of layering of network data, including the processes of encapsulation and de-encapsulation. When decoding traffic, both software and humans must know how to perform interpretation using fields and values that contain current or following protocol identification. There must be some designation of where protocols and fields begin and end. There may be an implicit unstated fixed length, a supplied variable length, or a computed length.

We examined the role of ARP to manage IPv4 to link layer addresses for local network communication. You learned that ARP is not secure and can be used to cause denial-of-service or MITM attacks. IPv6 has similar support and issues using Neighbor Discovery Protocol.

The next layer up on the TCP/IP model is IP. The function of IPv4 and IPv6 is to route a packet closer in hops to its ultimate destination. There are notable changes in IPv6, including a fixed-length header and extension headers.

We discovered that fragmentation is employed when a packet is larger than the MTU of a link it needs to traverse. All fragments have an identifier to associate them with other fragments in the same fragment train, an offset to designate their place among the others, and, finally, an MF flag to inform whether other fragments follow.

• 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 2 Bootcamp

BPF and TCP Connections

This section's bootcamp material is designed to reinforce the BPF filtering that was discussed earlier, in addition to refining your command-line skills. These skills will serve you well in the days to come, assisting you in slicing and dicing packets at the command line. This will prove to be very important when dealing with large packet captures or needing to rapidly identify records of interest in packet captures of any size.

After the lecture portion, we will work with some of the data from our capstone challenge. The relevant sections of questions will become active on the score server, and you will work through a series of challenges using the techniques discussed earlier and in this bootcamp material.

Leveraging Our Knowledge

Let's ask important questions based on our knowledge:

- How many services are running in the defended network?
- Are there services outside of the "Server" network? What are they?
- How many services are our users connecting to outside of the network?
- What kinds of ICMP activity are there? Are there indications of probing/scanning?

In the bootcamp associated with the first book, we learned the fundamentals of using command-line tools to parse and organize data. In this bootcamp, we will try to reinforce what we know already by putting it into practice to try to answer some important questions that would be relevant when attempting to investigate activity in a network or when being proactive in learning about our network during normal operations.

To answer the questions in the slide, we will draw not only on our command-line capabilities but also on our knowledge of protocols. As you will see, this isn't simply a matter of "Look for X," but instead we must think about what it is we'd like to find and what in the functioning of a protocol would reveal the answer.

Last Question First: ICMP Activity and Probing/Scanning

What kinds of ICMP activity are there?
Are there indications of probing/scanning?

Don't think facts; use what you know to predict

- Types and codes can tell us what kind of activity there is.
- Ping can be an indication of probing, but it's also used for heartbeats.
 - Could we identify different types of pings?
- What might these mean?
 - Unreachables (host, network, port)

As an example of applying our knowledge to know how to find the activity that we would like to identify, consider the ICMP question. On its face, it's simple to find ICMP packets. But what can those ICMP packets tell you?

Perhaps we could attempt to identify different ICMP types and codes that are in use. While an ICMP request or response could be a sign of a probe, it could also be a sign of some kind of a heartbeat (ensuring something is still up or reachable). While we might be able to identify various operating systems by looking at the payloads, other types of messages could show us other kinds of activities.

Think about what an unreachable message might mean. If it's a host unreachable, it could be a sign of a host that has gone down, or it could be an indicator of a network scan. On the other hand, if it's a network unreachable, it could be a sign that something's wrong with a route or an even stronger indication of scanning since the person doing so doesn't even know which networks are populated! A port unreachable, on the other hand, could be a sign that a service has failed or been turned off or a sign of a port scan of a host.

Looking at ICMP

Show one of each ICMP type in a capture file:

```
for i in $(seq 0 255); do tcpdump -r capture.pcap -n -c 1  
"icmp[0]=$i" 2>/dev/null | more; done
```

Look for things that aren't Windows pings:

```
tcpdump -r capture.pcap -n -X 'icmp[0]=8 and not  
icmp[8:4]=0x61626364'
```

Look for all unreachable:

```
tcpdump -r capture.pcap -n -X 'icmp[0]=3'
```

Consider each of the solutions to our questions presented on the slide. In the first, we use some simple Bash scripting, allowing us to interrogate a pcap file to show us one of each type of ICMP message within a capture file. This could reveal anomalous uses of ICMP or unusual types of ICMP errors that could be indicative of more unusual probes or scans.

The second example looks for the string “abcd” at offset 8 within an ICMP echo request (type 0). Microsoft Windows hosts will use the string “abcdefghijklmnopqrstuvwabcd...” as the majority of the payload for ICMP requests. Similarly, UNIX systems use a different ASCII pattern, which we could either add to our filter to exclude those as well, or which we could use to discriminate between Windows or UNIX and other systems... or other covert (or unusual) uses of the ICMP echo request/reply protocol.

The last example takes advantage of the groupings of ICMP messages. Since all unreachable messages are ICMP type 3 regardless of the specific cause, we can interrogate a capture file to identify all of these, possibly using that information to identify scanning or other malicious uses.

What About Services?

UDP services are tricky...

- We can't assume the service is running on a port less than 1024.
 - NFS, for instance, is typically on 2049.
- UDP doesn't require a response... It could be one way.

What if we looked at pairs of ports?

- Can you use cut to look at two fields at the same time?
- Could you find unique combinations of ports?
- From this, could you identify which port is most likely the service?
 - Which port numbers appear rarely? Rare ports are likely ephemeral.

Similarly, we can gain a great deal of insight about UDP services by reviewing the pcap files. UDP is a bit tricky when it come to identifying services, though, since UDP has no strong indicator regarding which host is the server and which is the client. We can't even assume that the low port is the server and the high port is the client. NFS, for instance, uses 2049 as the service port. ISAKMP usually performs its key exchange from port 500 to port 500, or possibly port 4500 to port 4500.

Still, we can figure it out because we understand how the protocol works. Using cut, we can extract two fields simultaneously. What if we extracted all pairs of UDP ports and then identified unique combinations, counting them? If NFS were in use, we would expect to find port 2049 in a number of the combinations, with more or less unique ports associated with each instance of 2049. Even if we didn't know that 2049 is typically NFS, we could readily identify port 2049 as a likely service since it appears in so many combinations!

TCP Services

TCP services are much more straightforward

- Looking for SYNs, we identify connections *to* a service.
 - We would be interested in cutting the *destination* port.
- Looking for SYN/ACKs, we identify *listening* ports where a service is running.
 - We would be interested in cutting the *source* ports.

We know how to find both SYN and SYN/ACK.

- We also know about other things that could conceal connections from us...
- Can you figure out what this refers to?

TCP services are much more straightforward to identify. Since TCP uses signaling with the TCP flags, we can easily determine which end of the connection is the server and which is the client, as long as we can see either the SYN or the SYN/ACK packet from that connection.

We can leverage this to target the kinds of data that we would like to find. For example, to find out which services our hosts are connecting to, we could look for SYN packets and extract the unique destination ports. On the other hand, if we are interested in finding out which services are listening, we would look for SYN/ACK packets, extracting the unique source ports.

Our knowledge of these protocols is fairly extensive at this point. There is another capability that might be implemented in our TCP stacks that could impact our identification of listening services in either direction. Can you figure out what it is without looking at the next page?

BPF and TCP Connections

Some hosts support ECN and some do not.

- We need to account for this when we are trying to answer certain questions.
- ECN negotiation occurs in the TCP flags or code bits.
- We can use bit masking to ignore the ECN bits!

ECN Flags									
Differentiated Services Bits in IP					ECN Bits in TCP				
Codepoint / Precedence Bits	Server	Client	CWR	ECN	URG	ACK	FIN	RST	SYN
SYN	0	0	1	1					
SYN/ACK	0	0	0	1					
ACK	0	1	0	0					
Normal, A->B	0	1	0	0					
Normal, B->A	1	0	0	0					
Congestion!	1	1	0	0					
Echo to sender	1	1	0	1					
Sender reduces window	1	1	1	1					

SEC503: Intrusion Detection In-Depth 72

SANS

SEC503: Intrusion Detection In-Depth 182

Remember that we discussed ECN negotiation. More specifically, we discovered that ECN negotiation will happen, from the perspective of TCP, within the 2 (previously reserved) high-order bits in the TCP flags or code bits, which are found in TCP offset 13. Failing to remember this can have a big impact on your analysis of packet data.

It is very common for an analyst to seek to answer the four questions mentioned previously:

- How many services are running in the defended network?
- Are there services outside of the “Server” network? What are they?
- How many services are our users connecting to outside of the network?
- What kinds of ICMP activity are there? Are there indications of probing/scanning?

Each of these questions will require us to use a BPF filter to look at 'tcp[13]' for various flag combinations involving SYN and ACK. The packet dissection novice will attempt to find SYN packets using 'tcp[13]=0x02', for example, but this is not a good plan.

ECN and BPF

- It is good to get into the habit of using BPF to select SYN and SYN/ACK packets:
 - `tcp[13]=0x12` is a *bad choice* for looking for SYN/ACK packets!
 - Remember that the two high-order bits in the high-order nibble of the TCP code bits are used for ECN.
- Better practice is to always mask those two bits off unless you are *specifically* looking for ECN negotiations:

`tcp[13] & 0x3f = 0x12`

Find all SYN/ACK, regardless of ECN

`tcp[13] & 0x3f = 0x02`

Find all SYN, regardless of ECN

Rather than looking at the entire byte, it is important to mask off the 2 high-order ECN bits, unless you are *specifically* looking for something ECN related. In other words, if you are attempting to find SYN packets, you will want to mask off the top 2 bits and see if what is left over contains 0x02 (`tcp[13] & 0x3f = 0x02`). If you are looking for SYN/ACK packets, you will, again, want to mask off the top 2 bits (`tcp[13] & 0x3f = 0x12`). By using this approach, you can identify SYN and SYN/ACK activity regardless of whether ECN is in use or not!

Section 2 Bootcamp Exercises!

Lab Workbook for Notes and Score Server for Questions

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.