SANS

www.sans.org

SECURITY 503
INTRUSION DETECTION
IN-DEPTH

503.2

Fundamentals of Traffic Analysis: Part II

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

SANS

www.sans.org

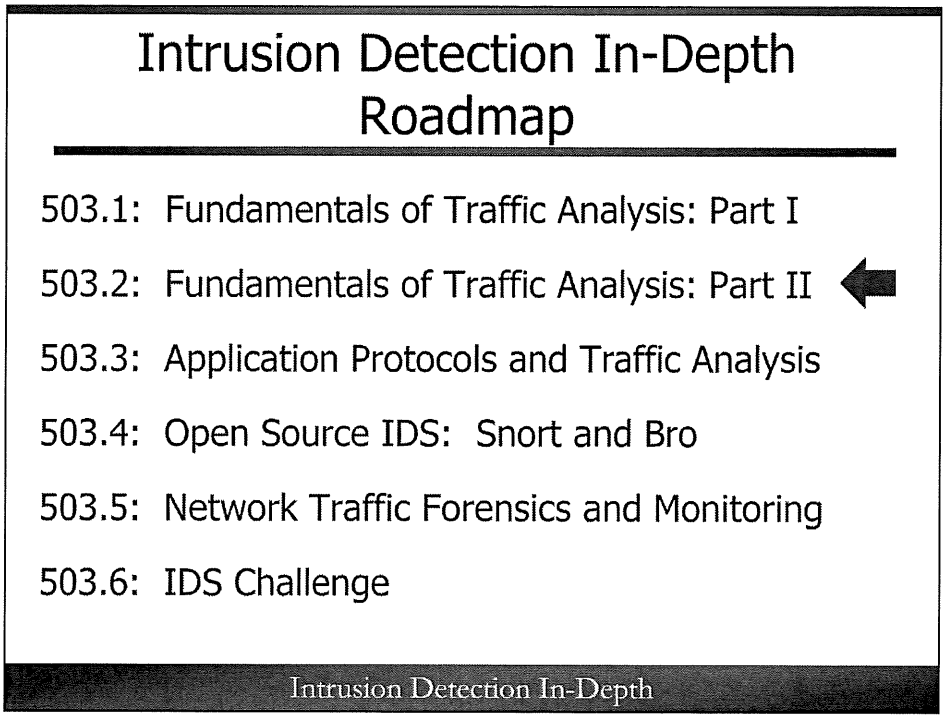
SECURITY 503
INTRUSION DETECTION
IN-DEPTH

503.2

Fundamentals of Traffic Analysis – Part 2

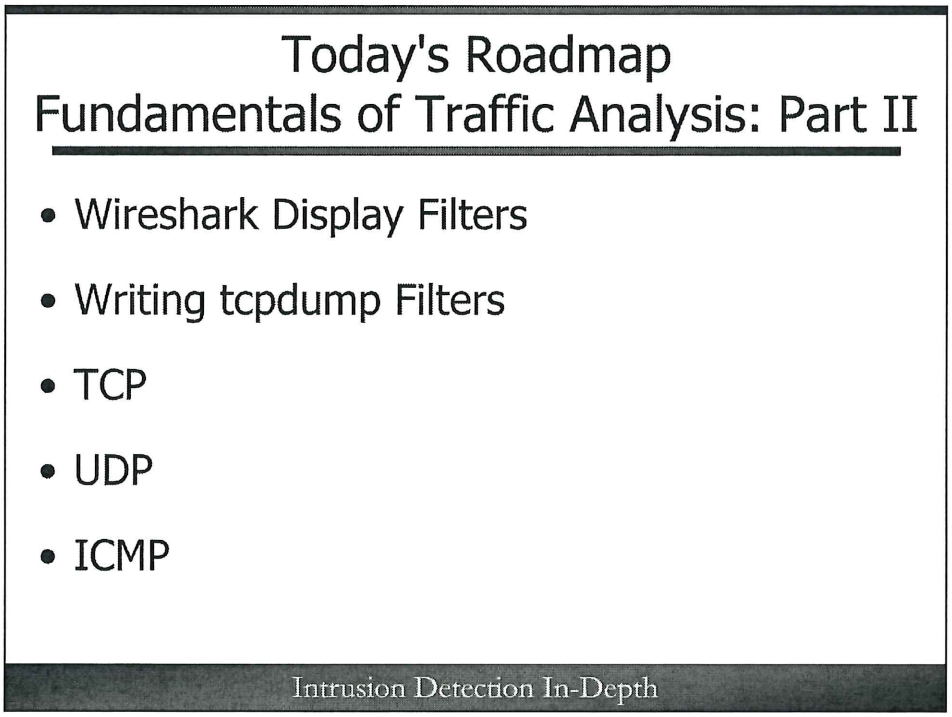
The right security training for your staff, at the right time, in the right location.

Sec503_2_A11_01



On this second day, we'll continue our tour up the TCP/IP model stack. We left you in suspense after Day 1, hanging from the intellectual cliff, no doubt with a sleepless night, wondering about what follows the IP layer. Today, we'll answer those very questions and quell your anxiety by examining protocols associated with the transport layer. Specifically, the topics will be TCP, UDP, and ICMP. It's much easier to examine these and the application layers that follow on Day 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 allow you to select what you want to view. Tcpdump uses Berkeley Packet Filters. Because Wireshark offers a more granular analysis of protocols, it has many more selection criteria to specify packets for viewing. Tcpdump BPF syntax is more esoteric and not nearly as flexible as Wireshark display filters. However, as we discussed in Day 1, tcpdump is very fast and efficient at selecting packets, especially on large pcaps. It is essential for an analyst to have masterful command of both tools.

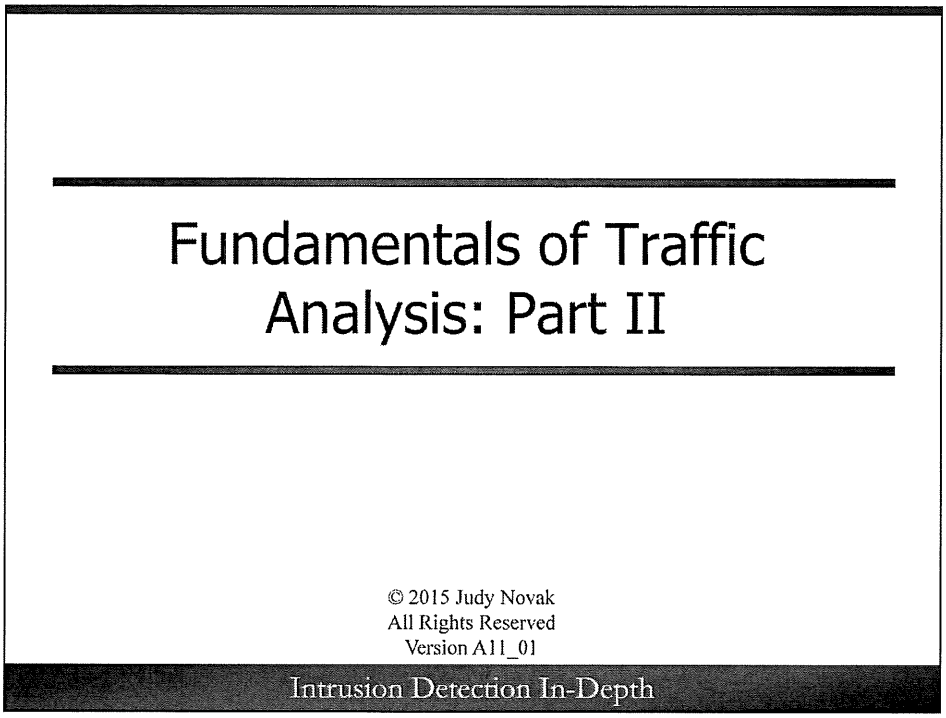


Here is a roadmap for Day 2. The day starts with more sophisticated ways to use the two main tools we discuss in this class – Wireshark and tcpdump. Many times when you are using these tools, you will want to find a particular record or two and perhaps sessions associated with those records. The means for specifying the criteria for record selection is known as Wireshark display filters or tcpdump Berkeley Packet Filters. As you'll soon see, Wireshark has an abundance of ways to select records and provide the selection criteria. Tcpdump filters are more limited, but ultimately more efficient.

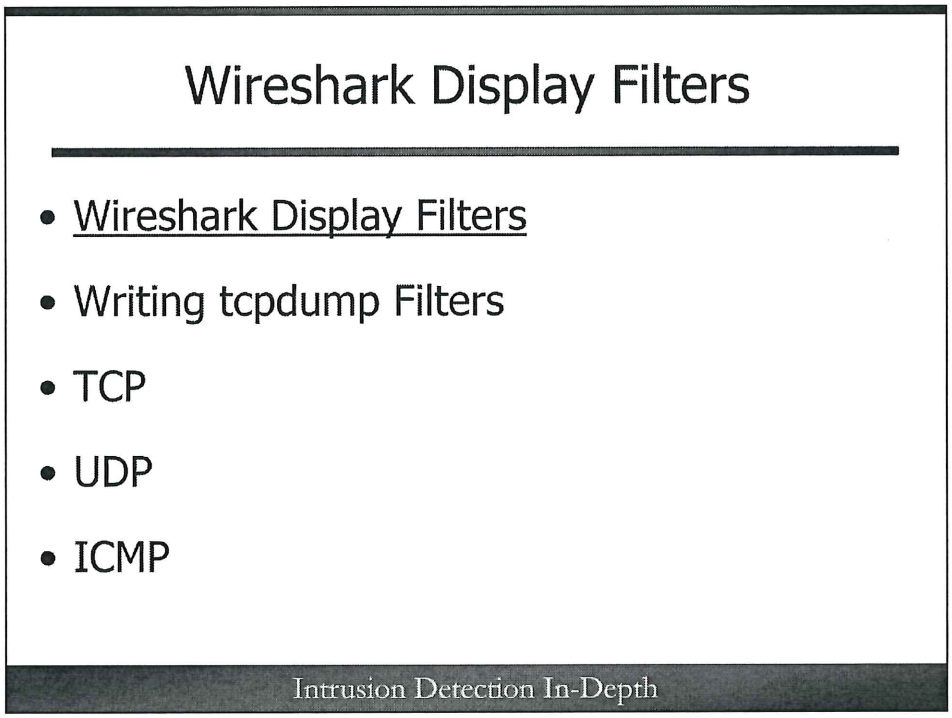
We'll continue with the discussion of layers as you go up the TCP/IP stack – namely the transport layer. TCP is a reliable transport layer, UDP – not so much, and ICMP is mostly used for error messages in IPv4 and error and informational messages in IPv6.

We will be covering many new tools and products today. We would like to cite the author or vendor of each in advance and give credit to them for their contributions.

Tool	Vendor/Author
traceroute	Van Jacobson, Steve Deering, Philip Wood, Tim Seaver, Ken Adelman
ping	Mike Muuss
LaBreat tarpit	Tom Liston
Smurf Attack	Tfreak
ptunnel	Daniel Stoedle
loki	daemon9 and alhambra



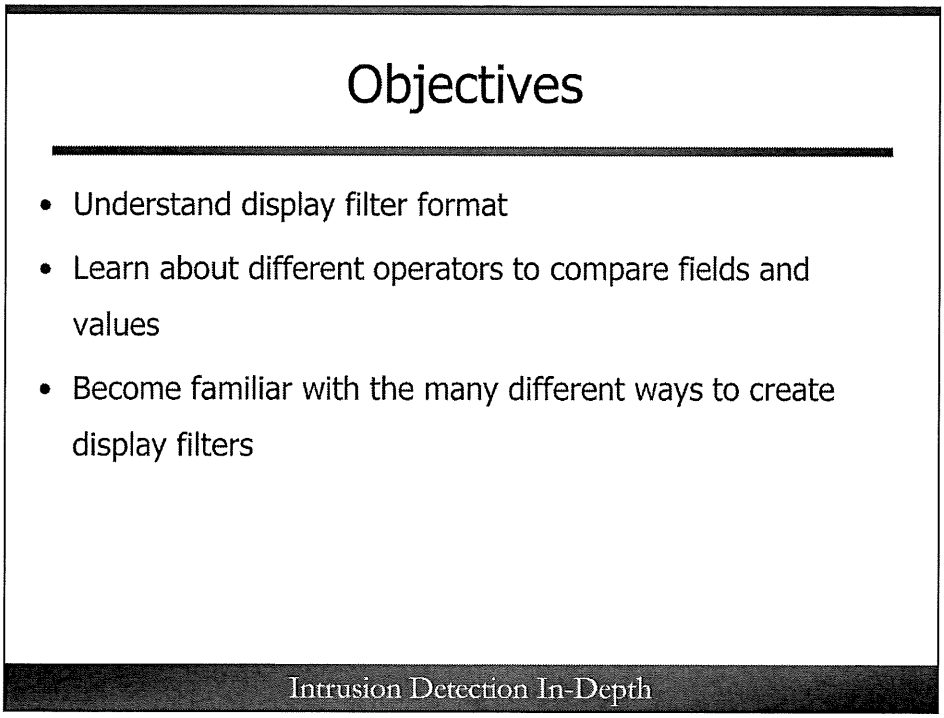
This page intentionally left blank.



One of the most powerful features of Wireshark is the ability 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 are of interest. Wireshark uses something known as a display filter to find one or more packets with a characteristic that you specify.

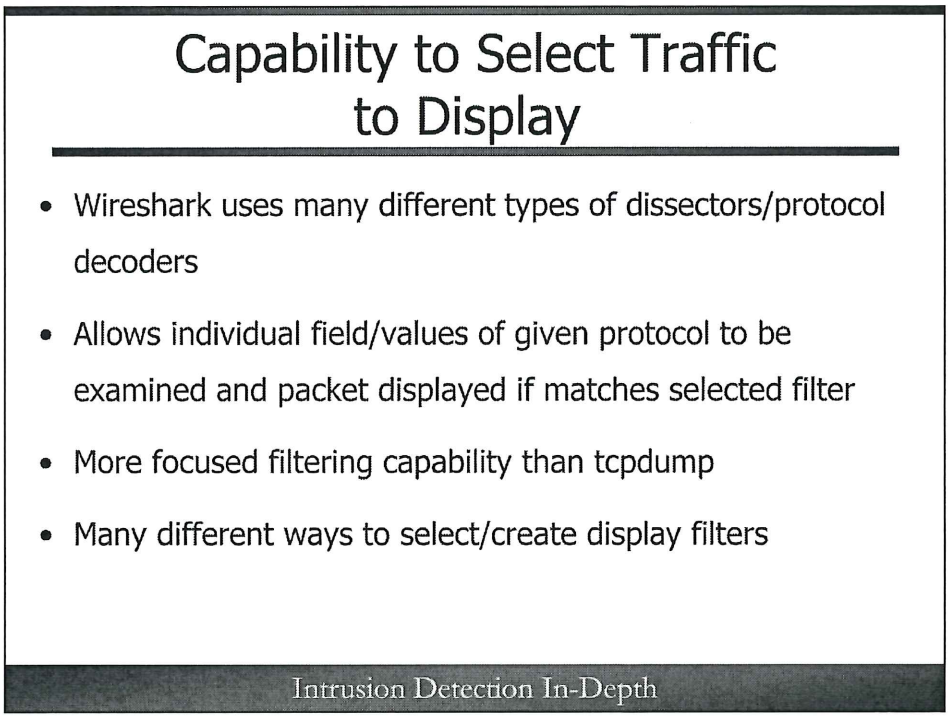
One of the most baffling aspects of learning Wireshark for me was understanding the distinction between a Wireshark capture filter and a display filter. I had a hard time with Wireshark at first because I thought the two were the same. But, that's not the case. Wireshark capture filters are the same as tcpdump Berkeley Packet Filters. Capture filters are used only upon traffic capture from the network. As you may know or will learn, BPF syntax is arcane, not particularly flexible nor user friendly. You have to figure out the offset into the packet of the field(s) you wish to filter upon and you have to perform bit masking for fields that are less than a byte in length. And you are restricted to selecting, at most, four consecutive bytes for comparison.

Wireshark display filters are totally different and much easier to use. 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 since it has some default commonly used display filters and it allows you to create your own 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.



First, you must know the format that is used for display filters. There may be several ways to identify a field or protocol that you want to examine. Also, there are different types of comparisons of fields and values. There can be the expected operators of equal, less than, etc., but there are some that can be used for text matches – such as "contains" and "matches".

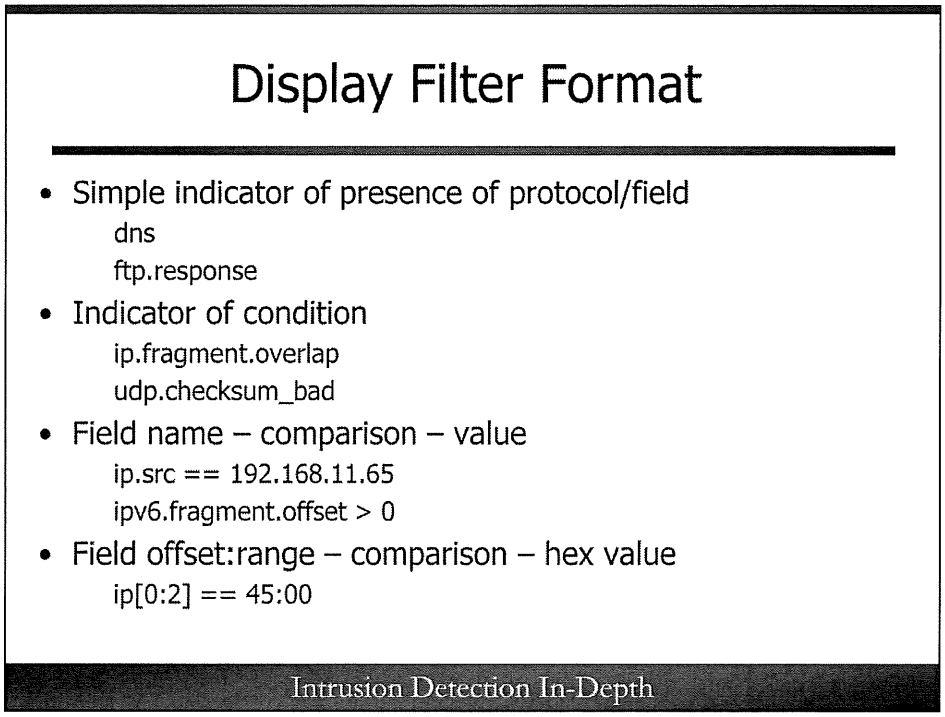
Wireshark makes it very easy to create display filters, supplying all kinds of assistance. Wireshark has a menu to list all the various protocols supported along with fields in those protocols available for display filter creation. You can use this as the starting point for display filter creation. Also, there are many different fields in Wireshark output that can be used to form a filter just by right clicking on the field and selecting menu options. For instance, say you are interested in seeing any traffic that has a certain Ethernet source address. You can right click on the output listing of that Ethernet source address and select appropriate options to apply the Ethernet source field and value as a filter. There are many more methods available for filter creation as you will soon see.



Wireshark has a very comprehensive set of protocol dissectors or decoders. This is one of the very impressive features that distinguishes Wireshark from many of the other open-source sniffers. A protocol 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 is able to parse all of the DNS resource records into components and show you the variable length fields like the IP/hostname or the DNS TTL value. Protocol dissection is the most accurate means of finding content when variable lengths are used in a given protocol.

Once a protocol is decoded, Wireshark then can expose both the fields and associated values to the user. This is a very 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, where 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, where low level is closer to the actual bits and bytes.



Display filters come in several formats depending upon what you are seeking. There are display filters that indicate the presence of a given protocol or field/value. For instance, the filter "dns" finds what Wireshark believes to be DNS records, based primarily on a port number of 53. The ftp.response filter looks for a source port associated with FTP.

Wireshark is also capable of doing some analysis of its own 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. In other words, the 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. Can do more complex filtering than tcpdump

A "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 since 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.

Finally, if there is a field/value combination that cannot be expressed using Wireshark filters, you can express it as an offset from the beginning of a protocol, the length in bytes, a comparison operator, and a strange notation for hexadecimal. We'll examine this more closely in an upcoming slide. The offset expression above looks for the hex value 0x45 00 in the first two bytes offset from the IP header. This could be expressed more easily with field names and values, but it is used simply as an instructive example of the format.

Comparison Operators

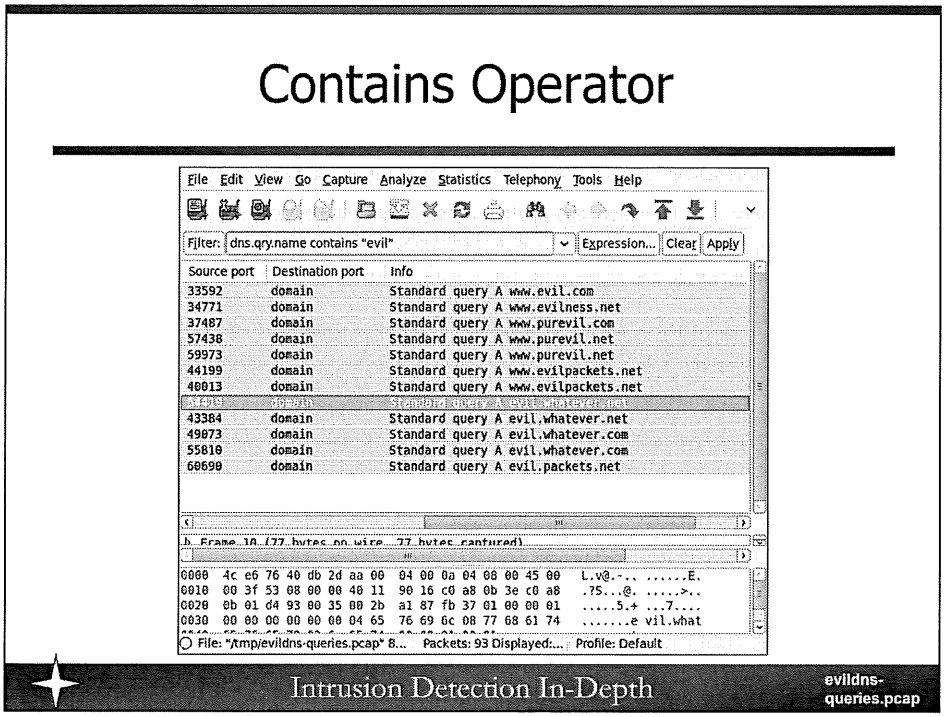
Comparison	Operator
Equal	==
Not equal	!=
Greater than	>
Less than	<
Greater than or equal	>=
Less than or equal	<=
Contains	contains
Regular expression	matches

Comparison	Operator
And	&&
Or	||
Not	!

Intrusion Detection In-Depth

Wireshark offers a typical set of comparison operations shown above. This is a sample of the most commonly used comparison operators; there are more than are listed that can be found in the documentation on the Wireshark website found at http://www.wireshark.org/docs. Look for the section on display filters in the Wireshark User's Guide.

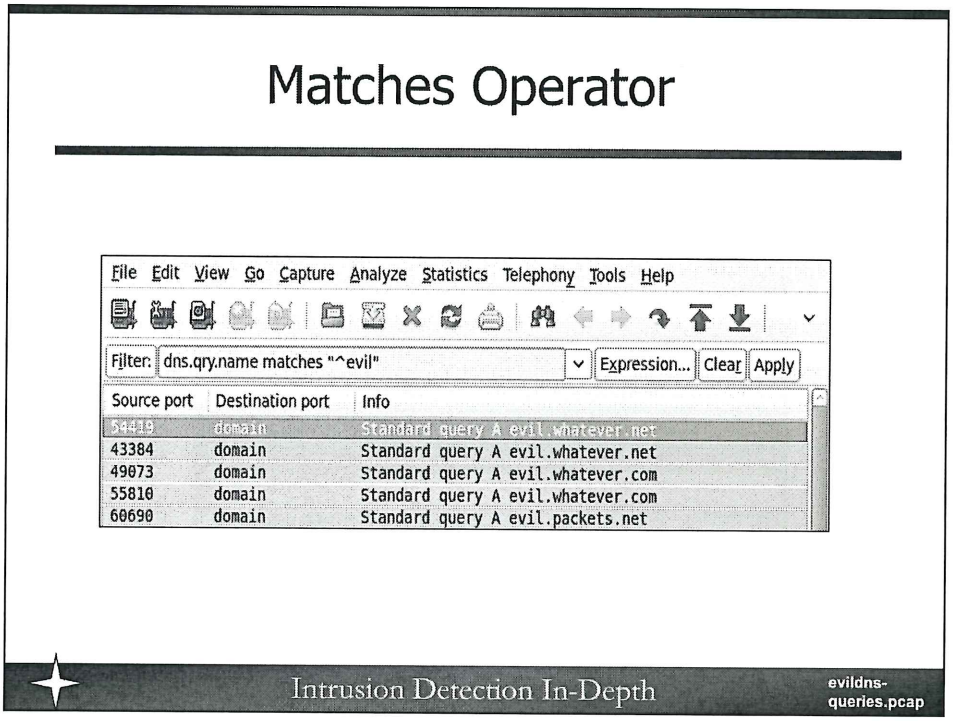
Most of these operators are straightforward and similar to expressions you may use in other languages. There are some that need more explanation in the following slides.



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

All of Day2 demonstration pcaps are found in /home/sans/demo-pcaps/Day2-demos on the VM.

To see the output, enter the following on the command line:
wireshark evildns-queries.pcap

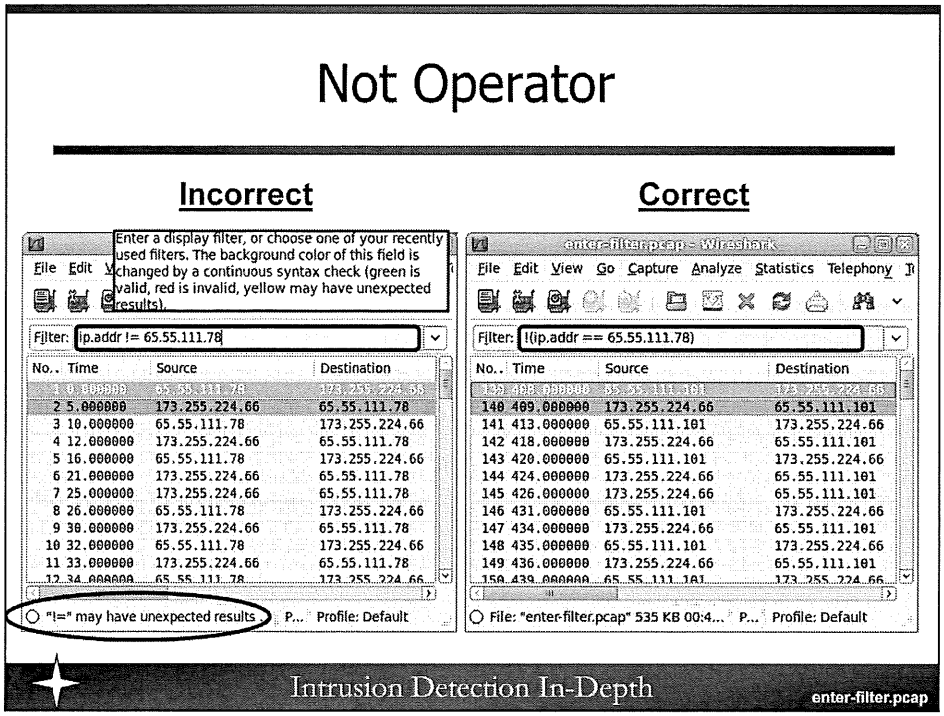


You're probably wondering what the difference is between the operators "contains" and "matches". The "matches" operator allows some basic regular expression syntax to be used to specify a more complex syntax of content to find. The filter above selects only records with "evil" at the beginning of the DNS query name. The "match" operator is not well documented. My advice is to experiment with regular expression syntax for matches that you'd like to find.

The "^" or carat is notation used in regular expressions to denote that what follows is found at the beginning of a line.

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

wireshark evildns-queries.pcap



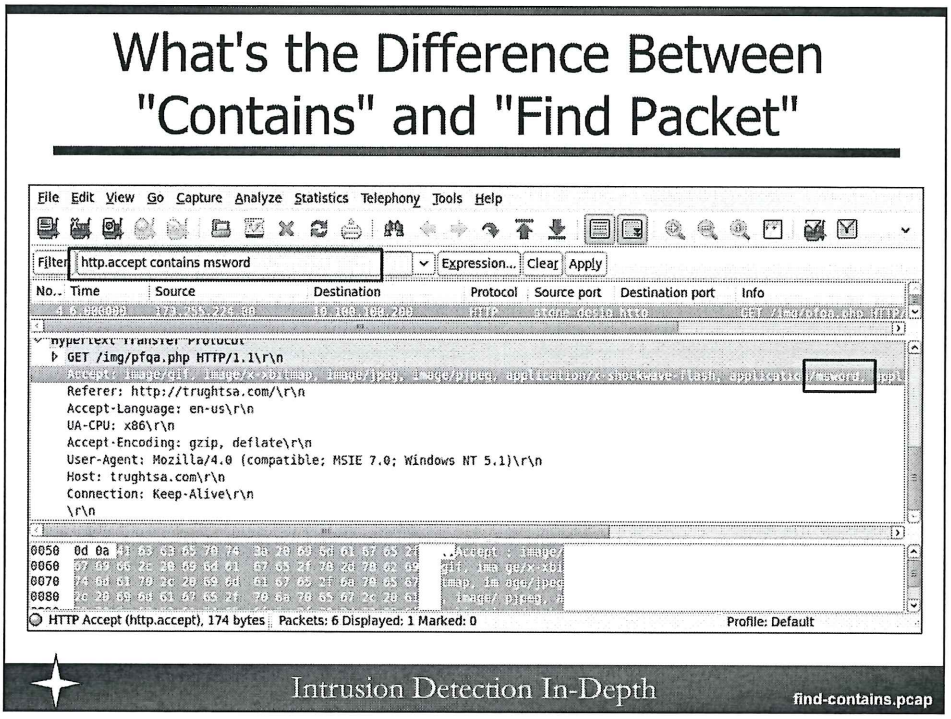
The "not" or "!" operator negates a particular condition. It seems straightforward, however one of the caveats offered in the documentation is the use of the "!=" operator. The example of "ip.addr != 65.55.111.78" is an incorrect filter to find all packets where the IP address is not 65.55.111.78. The 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.

This is probably a good time to digress a bit to discuss the background color in the filter input area. A white background appears if no filter is present. A green one appears when correct syntax is entered. Conversely, a red background appears if the syntax is incorrect. Finally, there is yellow background when Wireshark cautions you that the results may be unexpected. If you forget this, just hover over the filter input area and the guidance appears as above.

The proper way to compose the filter is "!(ip.addr == 65.55.111.78)" which is translated as no occurrences where the ip.addr field has a value of 65.55.111.78. Just be aware of this issue when you want to negate some value or expression, especially if it is a field found in both the source and destination packets.

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

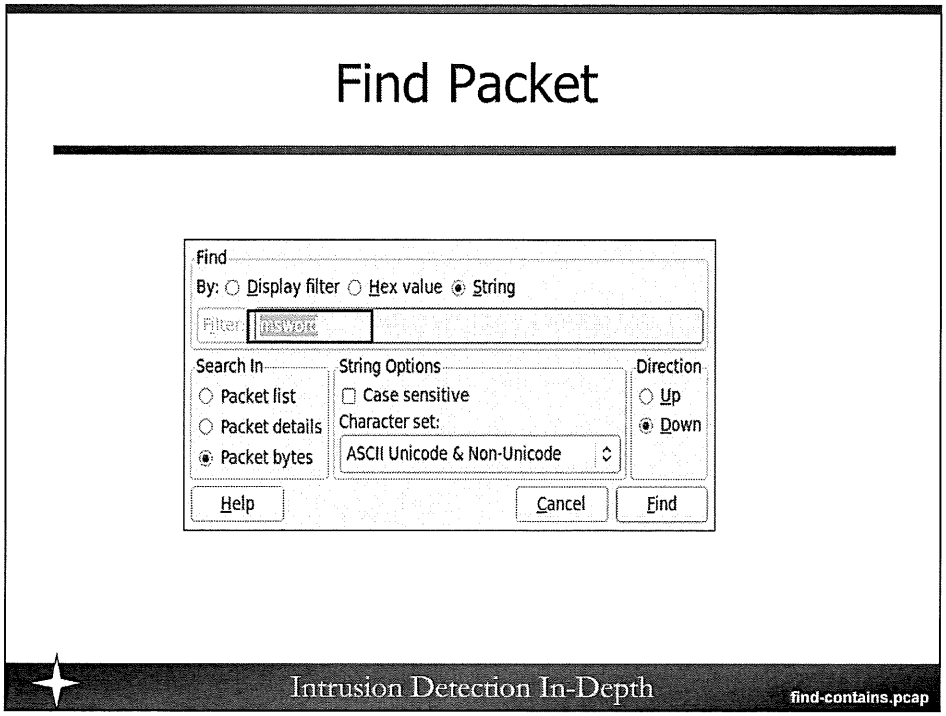
wireshark enter-filter.pcap



You may be wondering what the difference is between searching via the "contains" operator or by via the "Find Packet" option of the Edit menu. The "contains" operator allows you to do more focused granular searches on a particular field or protocol. For instance, above, we want to find "msword" only if it is in an HTTP Accept header. All discovered records will be displayed.

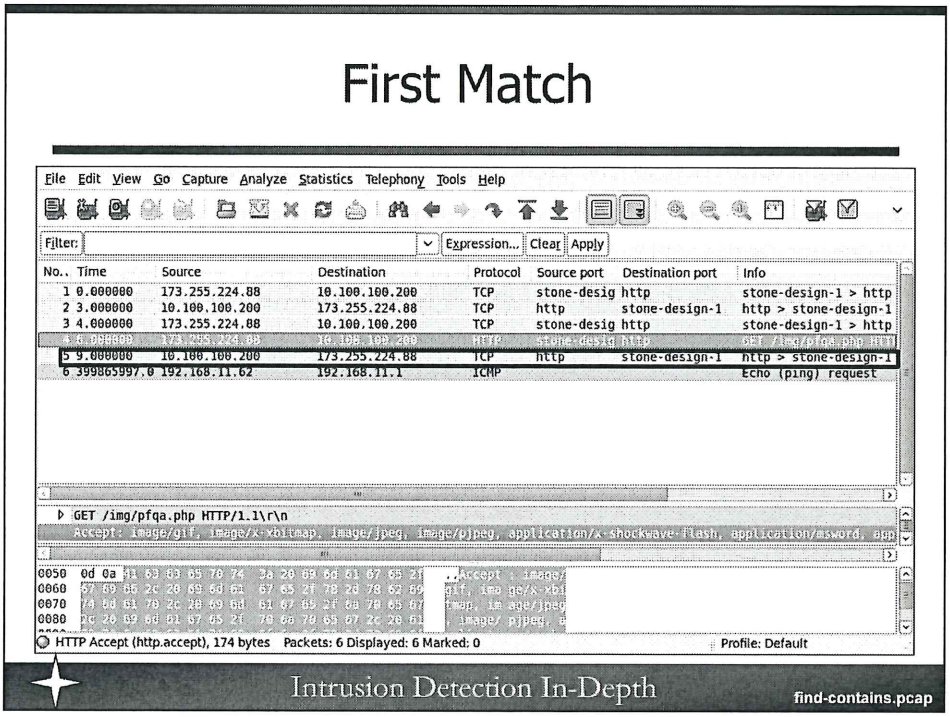
To see the output, enter the following on the command line:

wireshark find-contains.pcap



If you remember from the section "Introduction to Wireshark", the Edit menu has a "Find Packet..." option that you can use to find a packet based on some string or hex value that you select. The search is typically done on a value in the packet bytes, although it can be on the packet list or packet details. A match result highlights a record – leaving all other records displayed, while the "contains" results displays only matching records. You must do successive "Find Next" searches to find additional packets. As well, this is a more generic search because we are looking for the string value of "msword" anywhere in the packet payload, not just the HTTP Accept header.

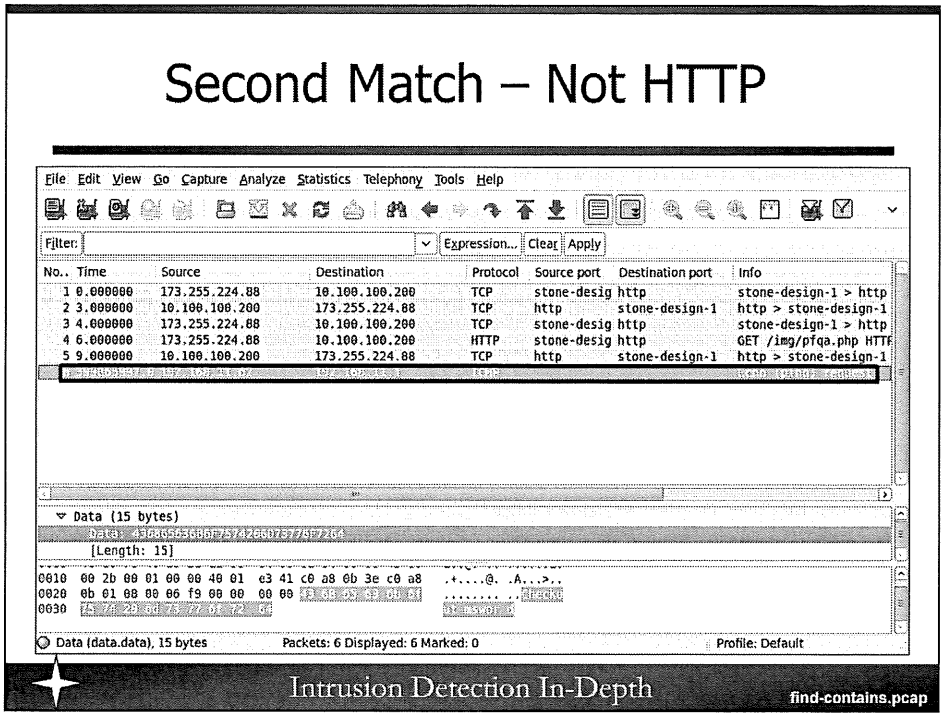
To see the output, enter the following on the command line:
wireshark find-contains.pcap



Record 4 contains the first match of the string "msword". It is in the HTTP Accept header as desired. This record is selected by either the "Find Packet" or "contains" searches that we performed.

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

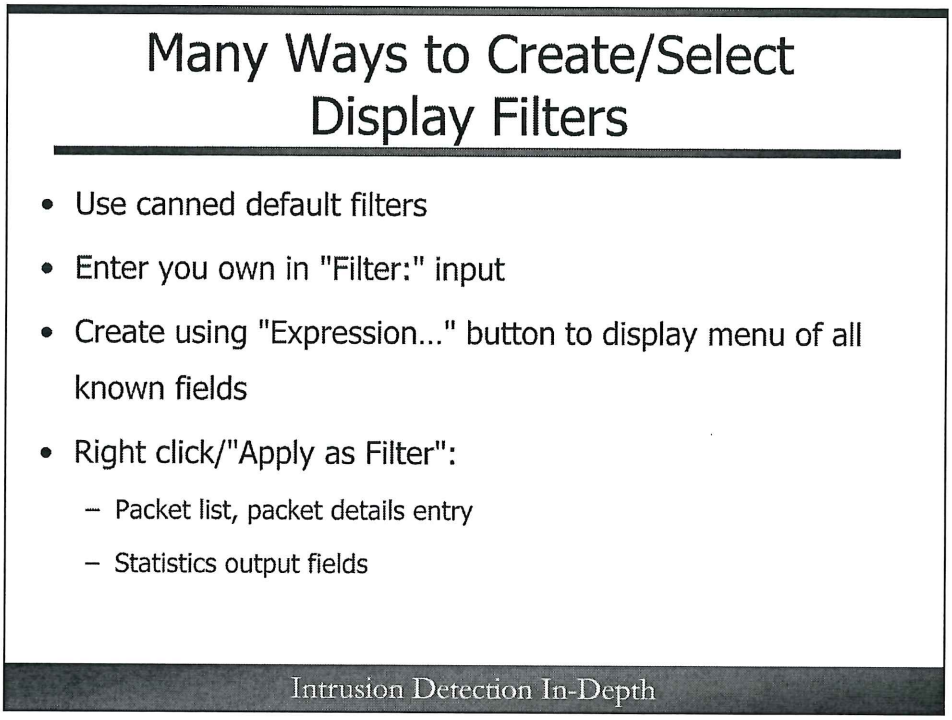
wireshark find-contains.pcap



The second match is an ICMP packet that has the string "msword" in the payload. Use the "Find Packet" when you want to examine each individual record separately that contains the match, yet you do not care about the context in the payload – just that the string exists in the payload. It's important that you select the correct option – "contains" or "Find Packet" - depending on your purpose for the search.

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

wireshark find-contains.pcap

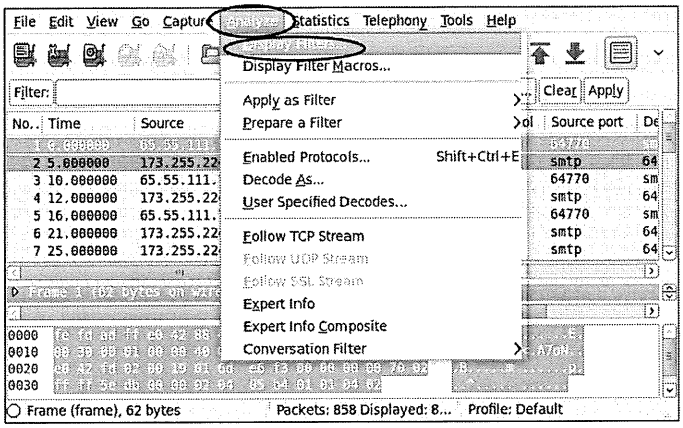


As mentioned, Wireshark offers you many different ways to create display filters. The type you create depends on your familiarity and comfort with Wireshark as well as what you want to accomplish. Wireshark comes with a small set of basic generic filters for some of the most common protocols. When you become more familiar with display filters, you may choose to create your own by entering one in the "Filter:" input field. You can get assistance, known as auto-complete where Wireshark attempts to help you complete your filter field name.

The "Expression" button allows you to display all Wireshark's supported protocols and fields within those protocols to select those you want and add comparison operators and values. Wireshark also offers a very clever feature of allowing you to right click on different displayed output to create a filter for a particular statistic, field and value or packet. Wireshark refers to this feature as "Apply as Filter". When used, Wireshark formats the filter entirely. Wireshark offers you menus of a set of conditions for whether or not you want to view or exclude records with the selected field and value.

The point is that Wireshark offers extensive methods for creating and selecting filters. All display filters require a given condition/field/packet to match, perhaps using an operator, and a value. Display filters can be as simple as selecting a given protocol – say ICMP. Or they can be very complex with multiple individual conditions combined with Boolean operators such as "and" or "or".

Selecting One of Wireshark's Canned Filters



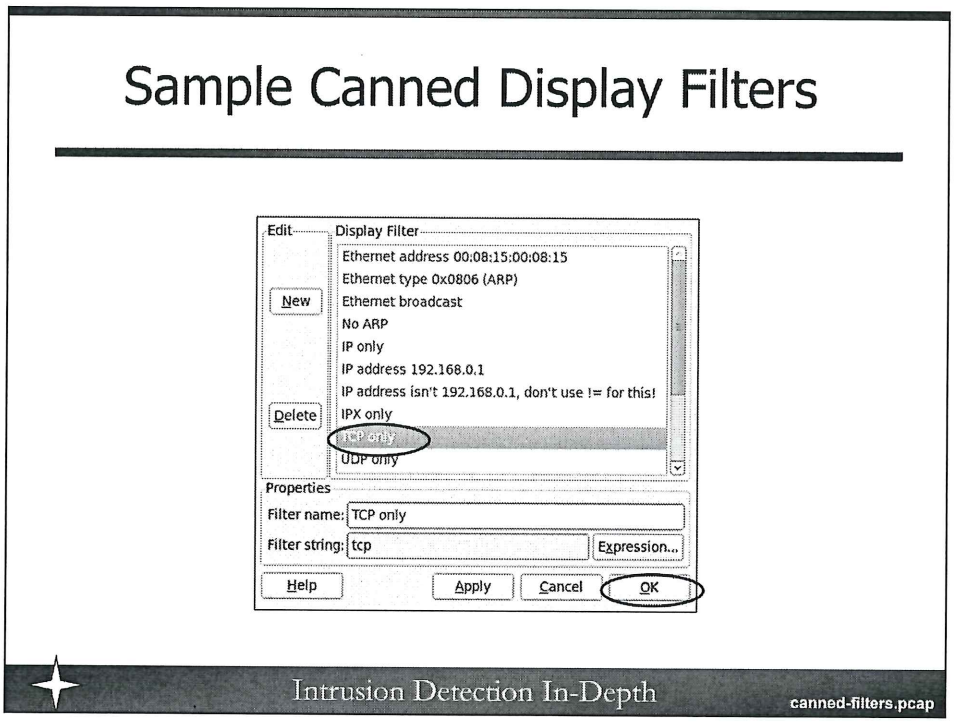
Intrusion Detection In-Depth

canned-filters.pcap

Some of the most common and simple display filters are available as default or canned filters. These can be accessed by navigating to Analyze→ Display Filters menu/selection or by clicking the Filter button to the left of the filter entry.

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

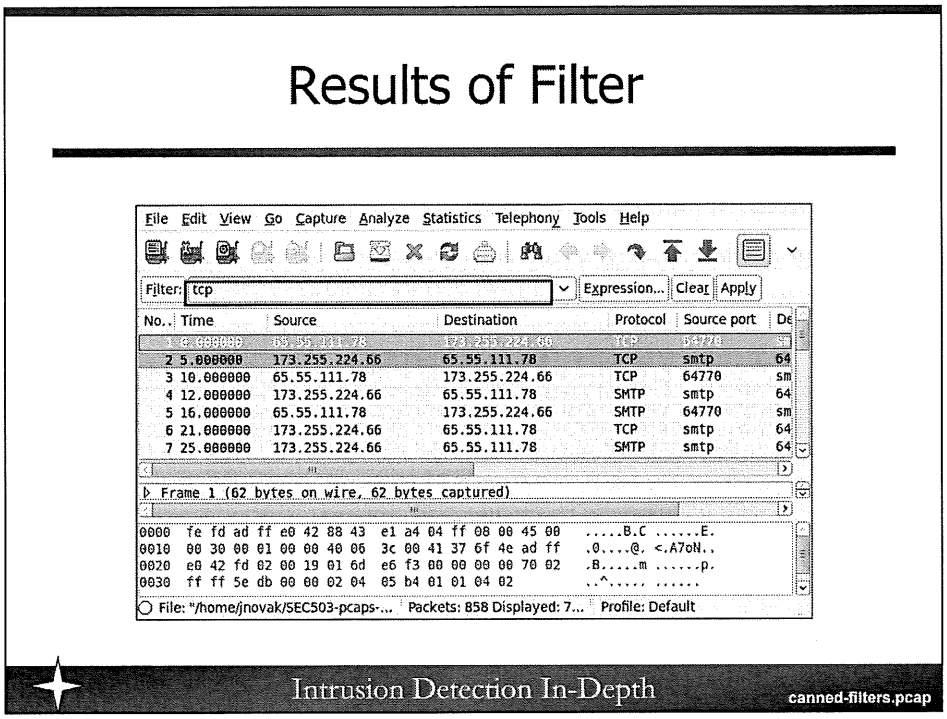
wireshark canned-filters.pcap



As you can see there are some very basic supplied filters. Suppose you wanted to see TCP traffic only. You would select the "TCP only" filter and select OK. The actual display filter that will appear in the Filter box after you select OK will be "tcp".

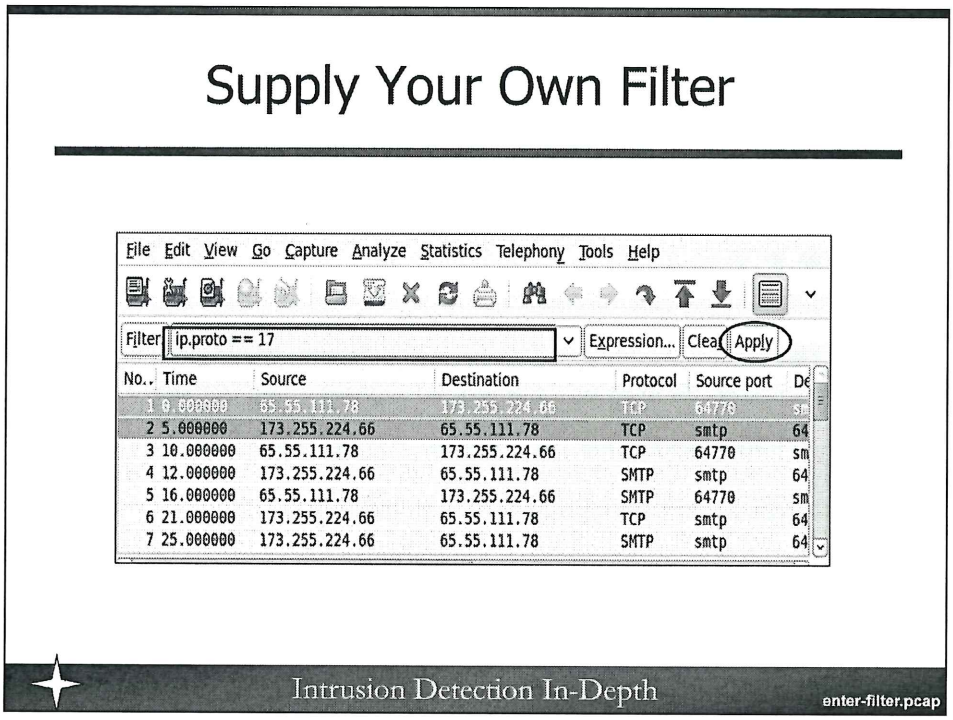
The filters that appear are contained in the file ./wireshark/dfilters found in the user's home directory.

To see the output, enter the following on the command line:
wireshark canned-filters.pcap



As you can see, the actual display filter is "tcp". This selects all the TCP records only.

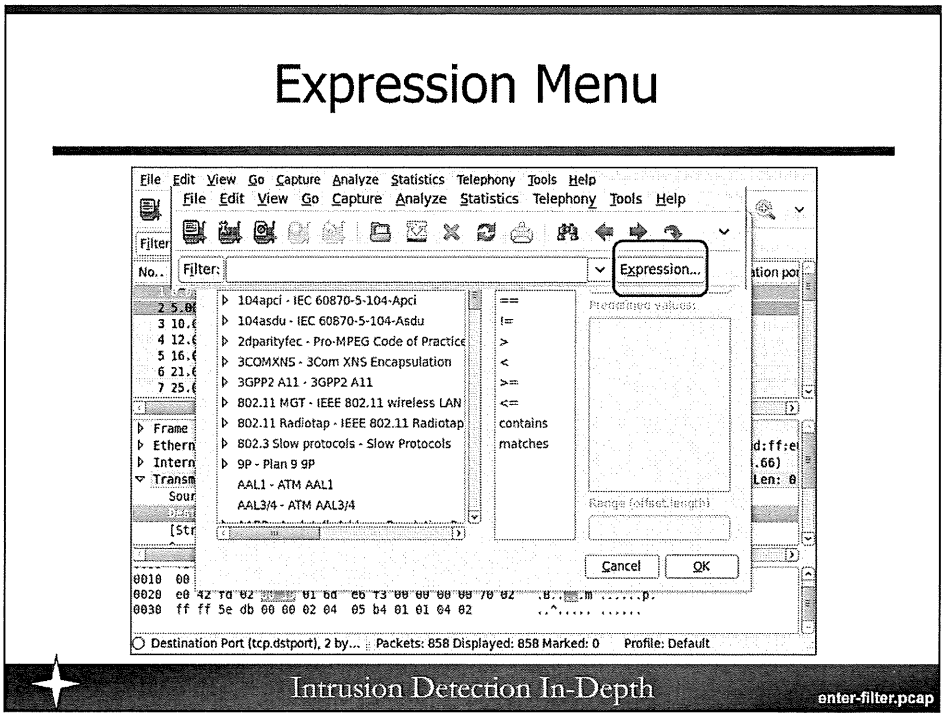
To see the output, enter the following on the command line:
wireshark canned-filters.pcap



Once you get 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 create a filter to look for protocol 17 – UDP. Once applied, only UDP records will be displayed.

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

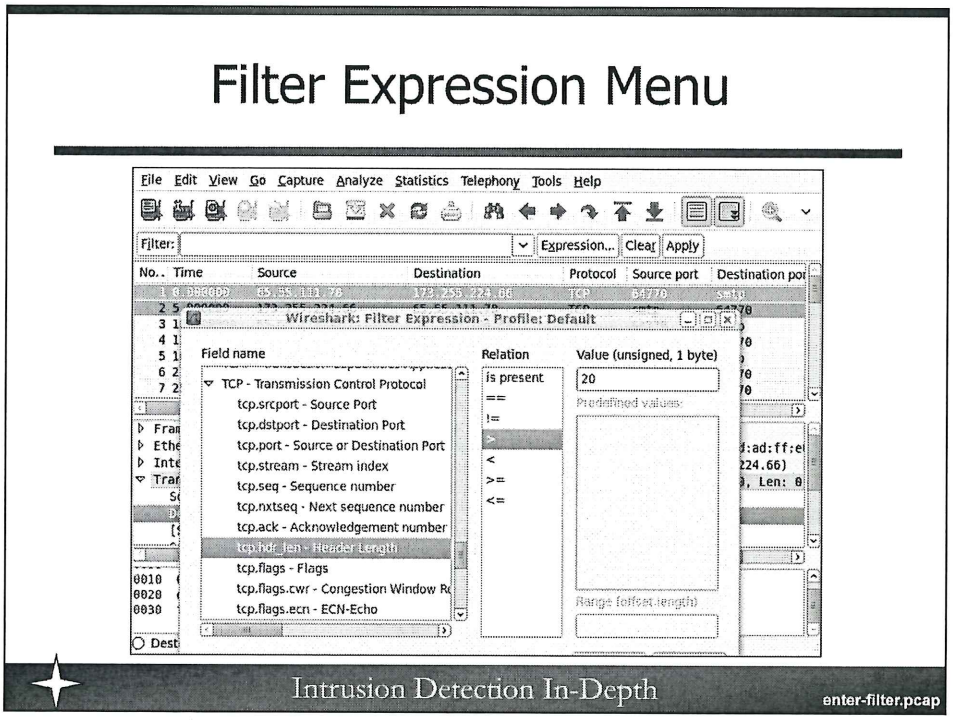
wireshark enter-filter.pcap



There is an "Expression" button to the right of the filter entry. When selected, a menu of many different types of protocols will appear. Each of these protocols can be expanded to reveal different fields for comparison or perhaps some conditions associated with the selected protocol. This menu shows you the available Wireshark dissectors and permits you to use and view the same fields and conditions that the dissectors do. This is a potent feature of Wireshark. You'll most often use the "Expression" button as a novice to Wireshark, or to explore the options for a given protocol, or you just have a lousy memory and cannot recall Wireshark syntax. But, wait - there's more ... Wireshark can help out for poor recall as we will see in the slide following the next one.

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

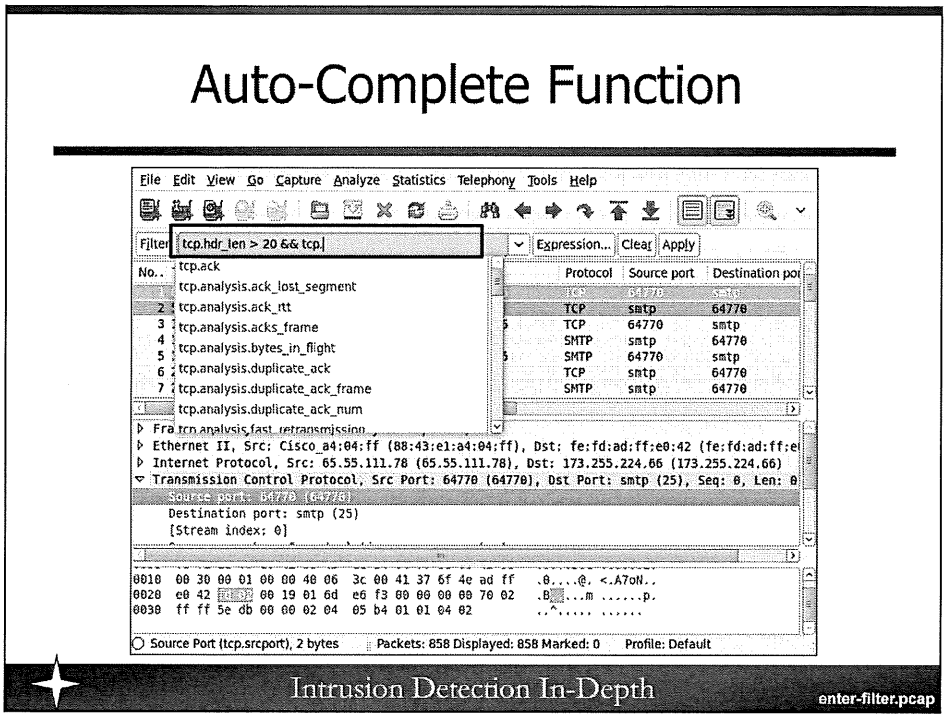
wireshark enter-filter.pcap



Once you find the field you'd like to search for a given value, Wireshark brings up a menu to help you select a relation and value. Note that Wireshark is cognizant of the appropriate relations that apply for the chosen field. For instance, we'd like to find any TCP packets that have a header length of greater than 20. Like the IP header, this indicates options. Wireshark knows that this is a numeric value. You are not presented relation options such as "contains" since that typically applies to string values.

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

wireshark enter-filter.pcap



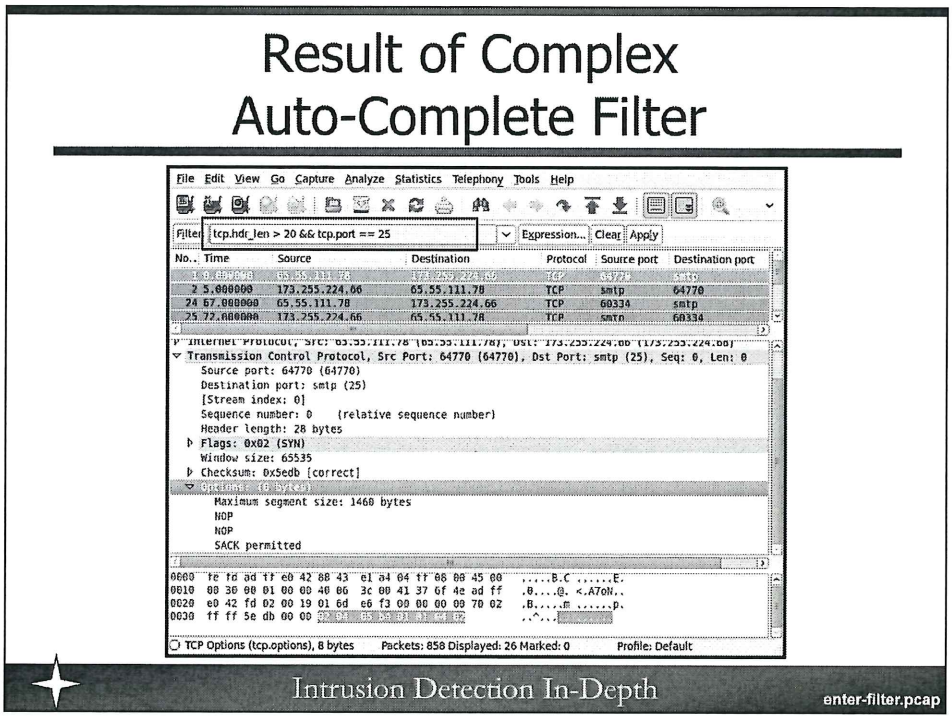
Wireshark has an amazing feature known as "auto-complete". Whenever you enter a filter on your own and select a particular protocol that you remember, but cannot remember a given field name for that protocol, simply type a period after the protocol. This is the catalyst for Wireshark to bring up all the like-named fields to 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" since that is the only field name or condition that it has with "tcp.h".

Let's expand on the previous filter to form a complex filter that contains two conditions. We enter "&&" in the filter since we'd like to limit the search to SMTP records – port 25. We enter "tcp." and a menu appears of all possibilities beginning with "tcp."

You have the choice to use "English" or "C-like" logical operators. For instance, while the "&&" was selected in this example, you could use "and" instead. An "or" operation can be expressed as "||" or "or" and the "not" operation as "!" or "not".

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

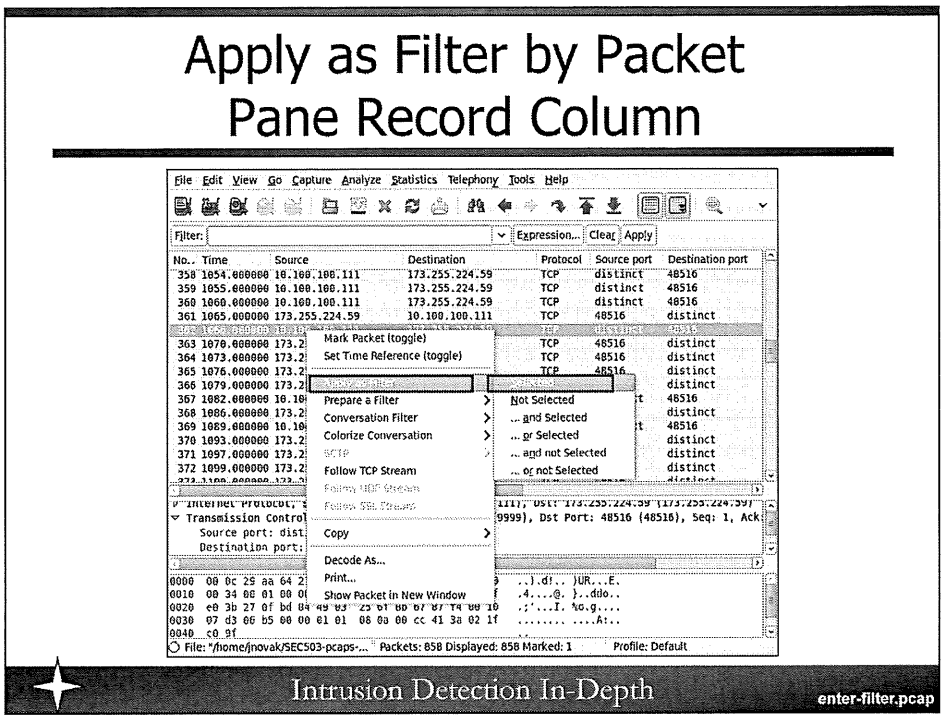
wireshark enter-filter.pcap



Finally, here is the result of our complex auto-complete assisted filter. This selects all SMTP records with TCP options. Wireshark assisted us with selecting the ".port" notation while we supplied the "= = 25" condition.

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

wireshark enter-filter.pcap

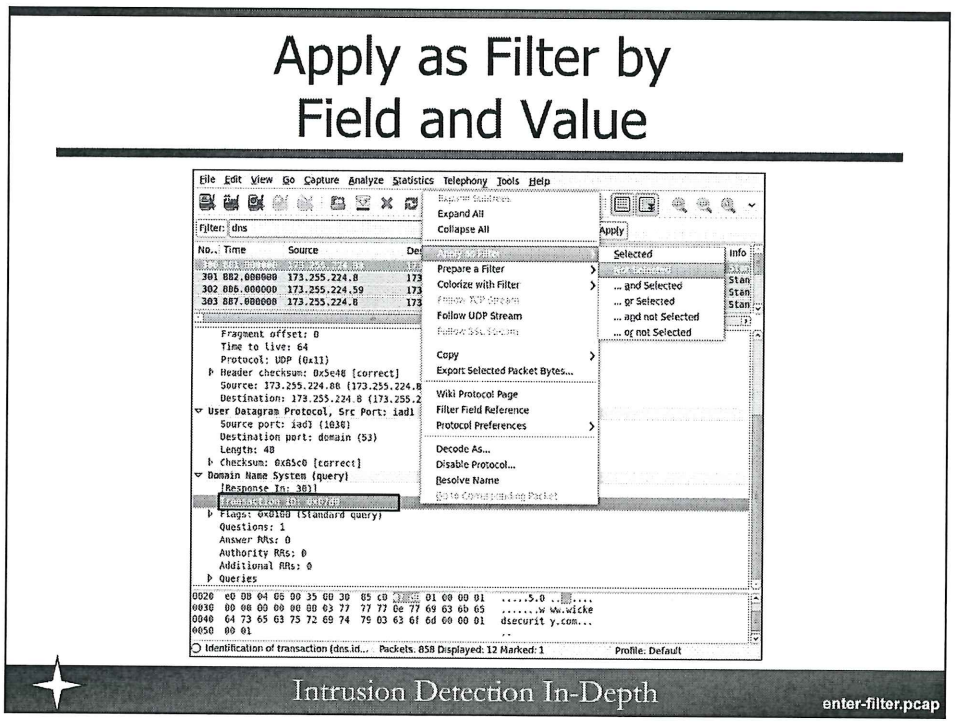


Another feature is the "Apply as Filter" menu option that appears when you right click on a record or a particular field/and value. An "Apply as Filter" on a record allows you to create a simple or complex condition to select or not select based on the column in the record where your cursor is when you right click.

We navigate to record 362 and wish to find all records with a similar source IP address of 10.100.100.111. Place the cursor under the Source column, right click, and a menu appears with an option of "Apply as Filter". There is also a "Prepare as Filter" option that allows you to create a complex filter yet not immediately apply it. Select the "Apply as Filter" option and a second menu appears. This menu presents many different options for including or excluding records based on the condition you've chosen – in this case, a source IP of 10.100.100.111. We choose the "Selected" option to include those records.

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

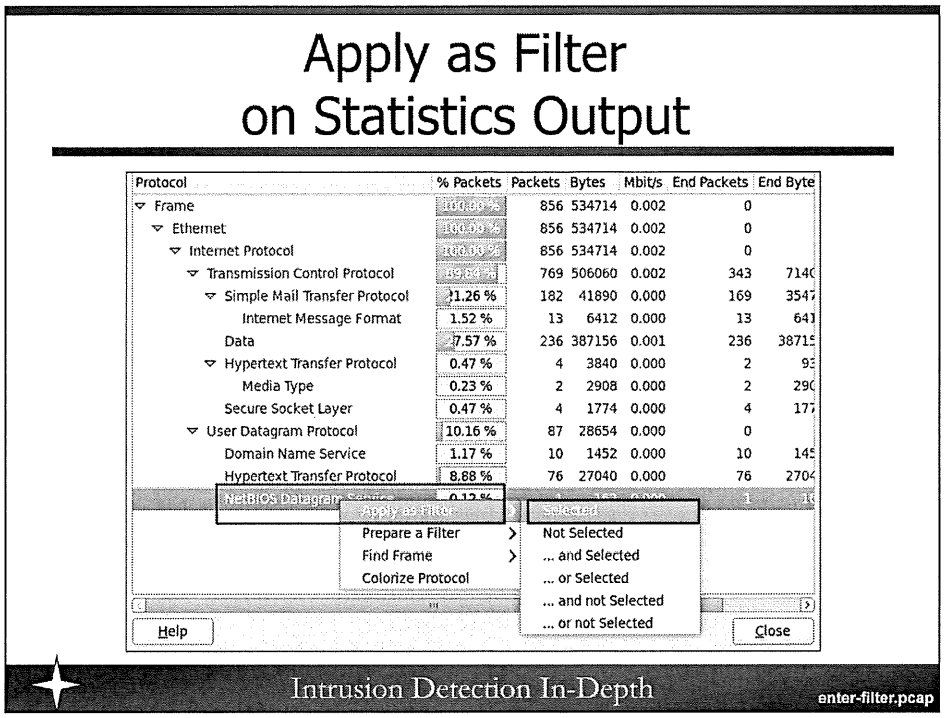
wireshark enter-filter.pcap



This time we bring up the "Apply as Filter" menu selection as the cursor is placed on the Transaction ID of a DNS record. We have opted to view all records that do not have a DNS Transaction ID of 0x07d0 as we choose the "Not Selected" menu option.

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

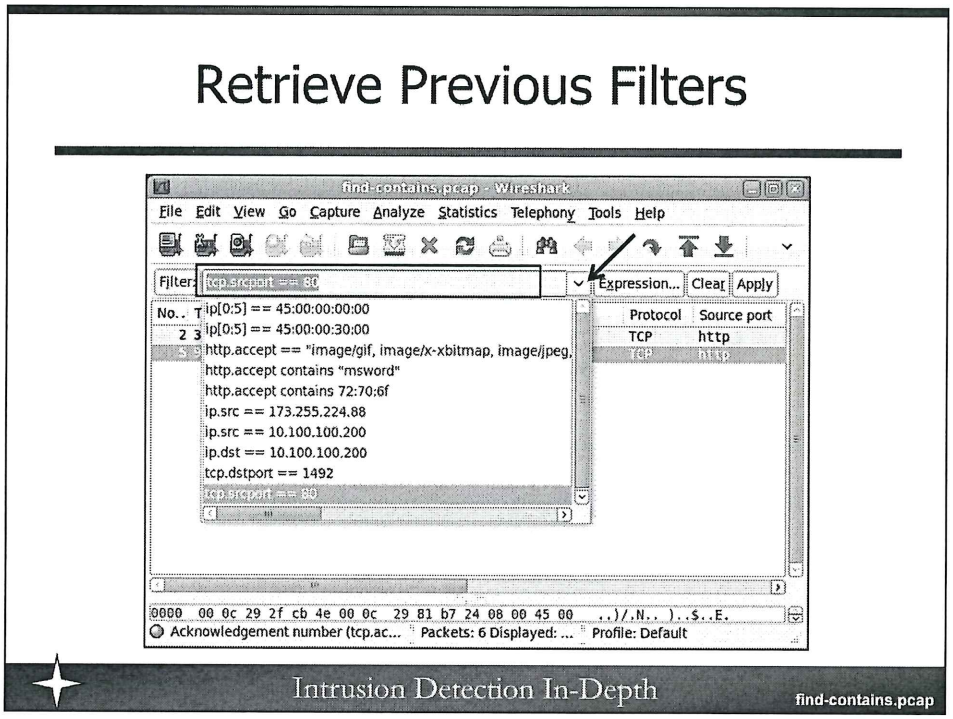
wireshark enter-filter.pcap



In case you don't have enough "Apply as Filter" options, you can also use many of the statistics output entries to begin your search. In this particular case, all NetBIOS records are selected from the Protocol Hierarchy Statistics menu.

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

wireshark enter-filter.pcap

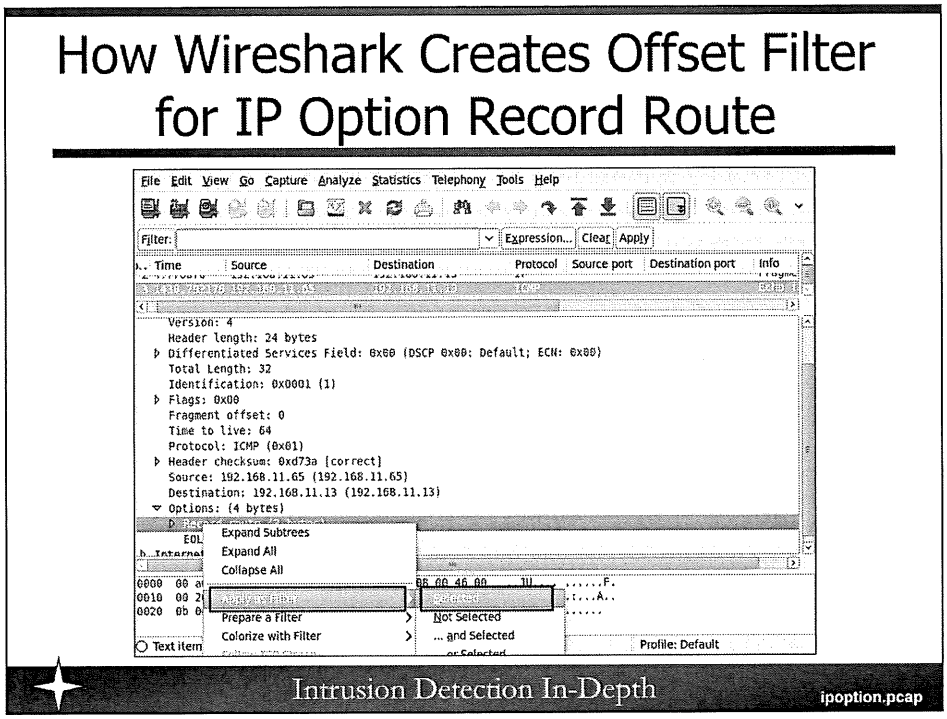


A time-saving feature is the capability to retrieve some of the recent filters that you've used. This is done by clicking on the down arrow next to the filter entry. Click on the filter that you want to use and it appears in the filter entry field.

These display filters are stored in .wireshark/recent_common file associated with the Wireshark user. You'll find recently used pcap files listed in there too. 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.

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

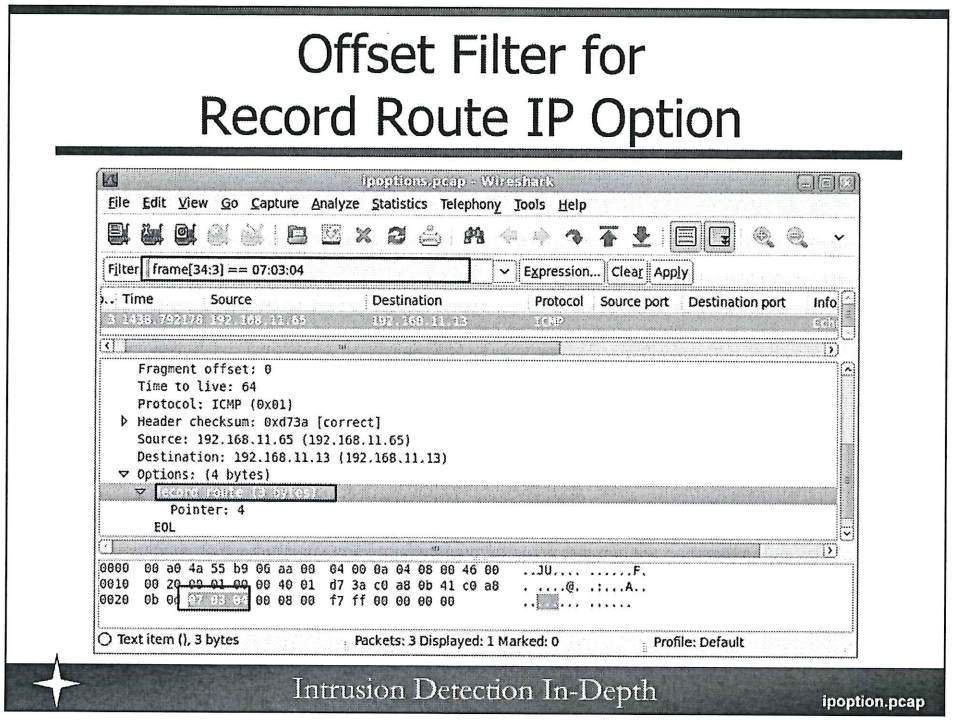
wireshark find-contains.pcap



Believe it or not there are some fields for which there are no Wireshark filters. That's hard to imagine with all the choices available to you. For instance, suppose you want to find all records that have the IP option record route. There is no filter for this. So, let's see how Wireshark finds it when an Apply as Filter is performed on the field of a record that has an IP option record route.

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

wireshark ipoption.pcap



What appears in the filter field is kind of disturbing looking! The syntax that you see, at least as far as designating the offset looks, very similar to BPF syntax that you will learn in the next section. The first part consists of the protocol that is used as the offset beginning. Wireshark elected to use the beginning of the frame, though it really could have used the beginning of the IP header as a more logical start point of an IP option.

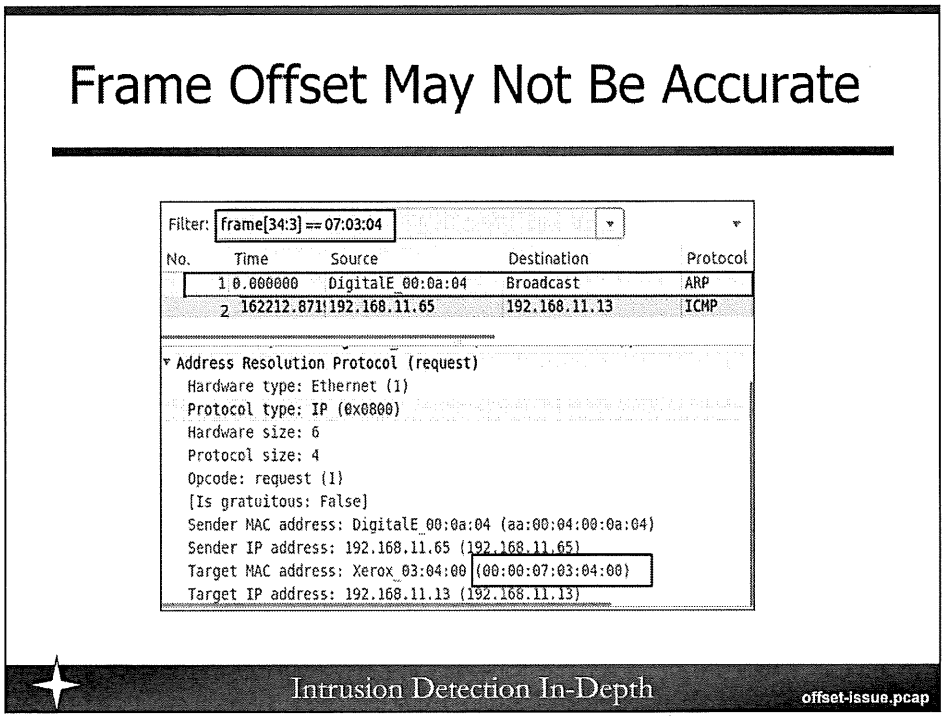
Regardless, Wireshark begins the filter match at 34 bytes offset from the beginning of the frame and looks for the next 3 bytes. Again, BPF has a similar format, yet while Wireshark does not have a maximum number of contiguous bytes for matching, while BPF is limited to 4. That takes care of the expressing the match field. Next we see a familiar operator of "= =".

Now what's that strange notation of "07:03:04"? That is Wireshark's notation for 0x07 03 04. It would have been a lot more user friendly, coherent, and cooperative had Wireshark used the same notation for hexadecimal as most every other tool in existence on the entire planet. Hey, no bitterness here! This is a small inconvenience given everything Wireshark offers – yet it would have been nicer to use the standard format for hexadecimal. The bytes 0x07 03 04 represent the IP option number of 7 for record router, the record route option as a length of 3 and the record route option pointer is 4.

The point is that Wireshark may not always behave the way you expect it to. It is an excellent tool, but not without nuances or anomalies.

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

wireshark ipoption.pcap



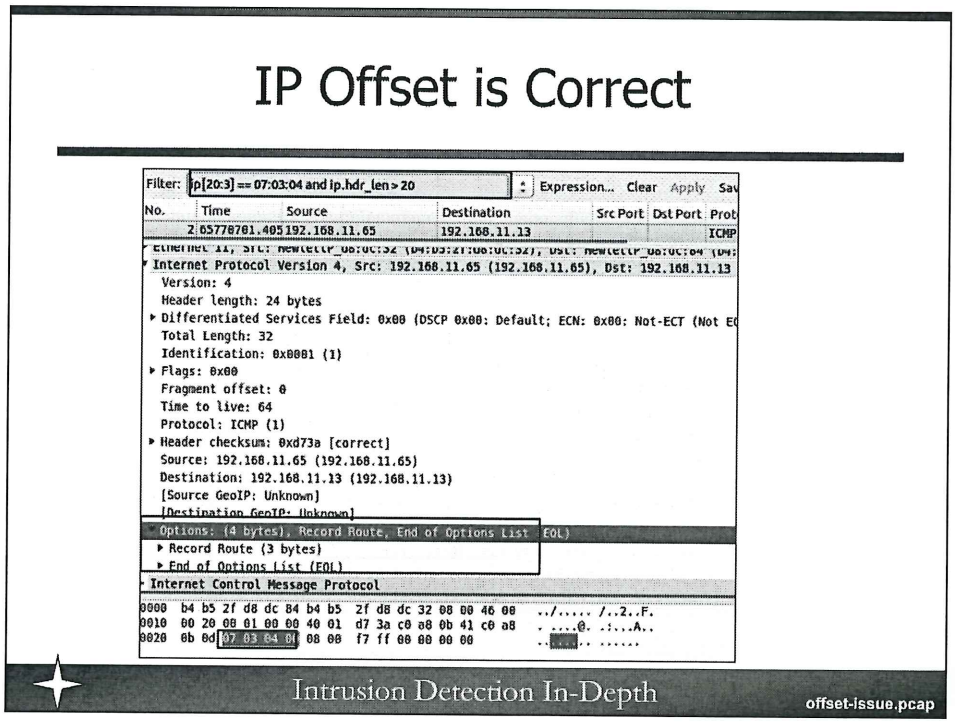
To test the theory that using a frame offset may not be the best way to search for IP options, an ARP packet was crafted that has 0x07 03 04 beginning at the 20th byte offset of the ARP header – or 34 bytes offset from the frame header. This was accomplished by assigning a target MAC address of "00:00:07:03:04:00". Fortunately, the field that spans the 20-22nd bytes inclusive accepts these values as legitimate ones since they fall within a MAC address. This packet was merged with the file ipoptions.pcap shown in Wireshark on the previous slide.

When this same filter that uses the beginning of the frame as the starting point for the offset is applied to the new pcap, you can see that it now includes the ARP packet containing 0x07 03 04 at the 34th byte offset from the frame. As we discussed in Day 1, ARP is a distinct protocol that follows the frame header, just as IPv4 or IPv6 are appropriate layers to follow the frame header.

The filter that Wireshark created when filter on IP options is not correct. It disregards the protocol layer we selected, namely IP. Let's fix this on the next slide.

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

wireshark offset-issue.pcap



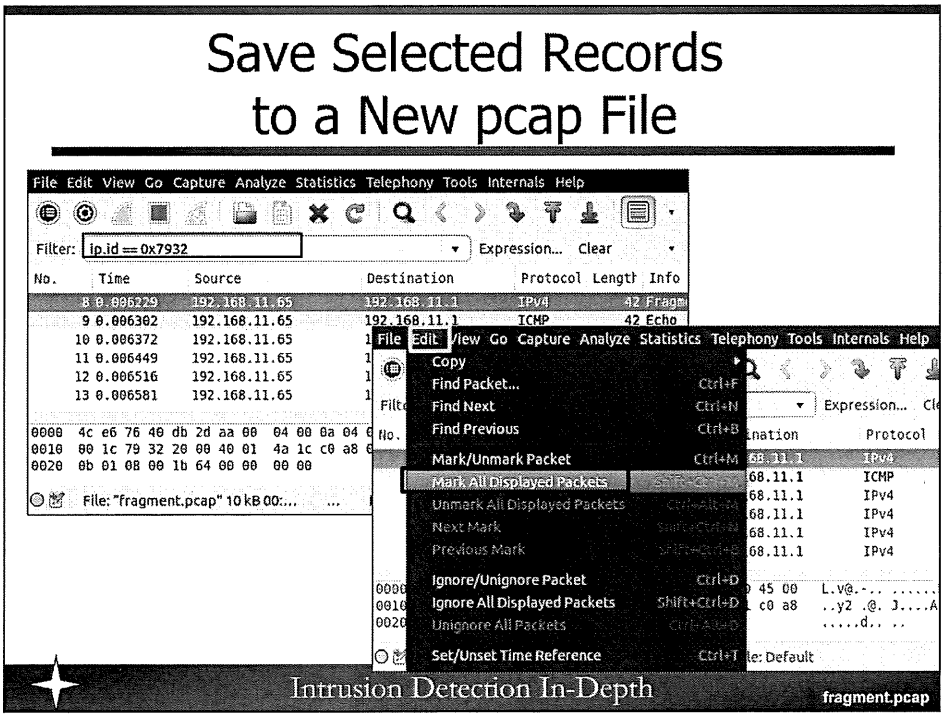
A filter that designates an offset of 20 from the beginning of the IP header where the IP header length is greater than 20 bytes selects the single record with IP options. A filter of "ip[20:3] = = 07:03:04 and ip.hdr_len > 20" looks 20 bytes offset from the beginning of the IP header and finds the record route IP option. It is not fooled by the ARP packet.

There are a couple of lessons from this examination. First, as mentioned before, Wireshark is a most excellent tool, albeit not perfect. With all the functionality that it provides, you would imagine that something might not be precise. This highlights our need to understand the protocol layering – Ethernet frame, followed by either IP or ARP. Finding IP options using Wireshark is no easier than using tcpdump for that matter. Both require offsets – Wireshark selects an imperfect layer to use while you as the user and creator of a filter for tcpdump need to know that you must start 20 bytes offset from the beginning of the IP header. We once again revisit a recurring theme that the tools you employ make their own interpretations of traffic; therefore it is necessary for you to be able to do your own should you come across a situation such as this.

Second, this emphasizes the necessity of beginning offset counting at the appropriate layer. If you are looking for IP options and need to designate an offset – start counting from the IP header. If you want to look for a value in TCP that requires an offset, begin counting at the TCP header.

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

wireshark offset-issue.pcap

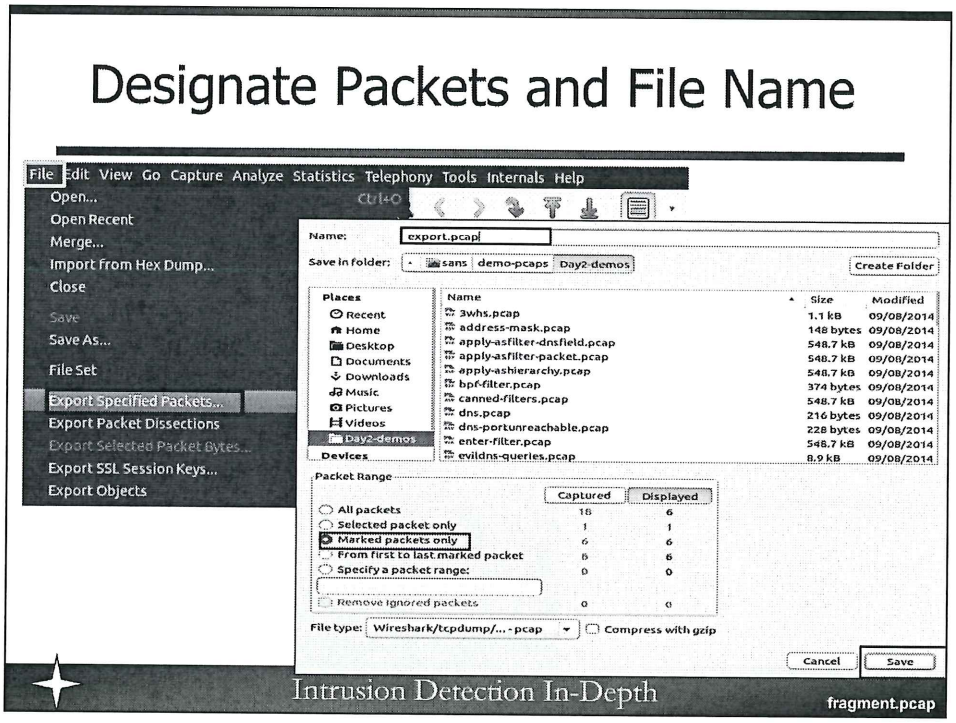


There have been times when I wanted to create a new tcpdump pcap file where I either select or omit records from a current pcap file. Sometimes it is easy to specify a tcpdump BPF field and value – other times not. But, Wireshark allows you to do this with relative ease. For instance suppose you want to save records with IP ID number of 31026 only. We've already performed an "Apply as Filter" operation on that field and all matching records are selected.

First mark all packets using the Edit → Mark All Displayed Packets option as seen in the left upper display. After this is complete you should see all the packets highlighted as in the right lower display.

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

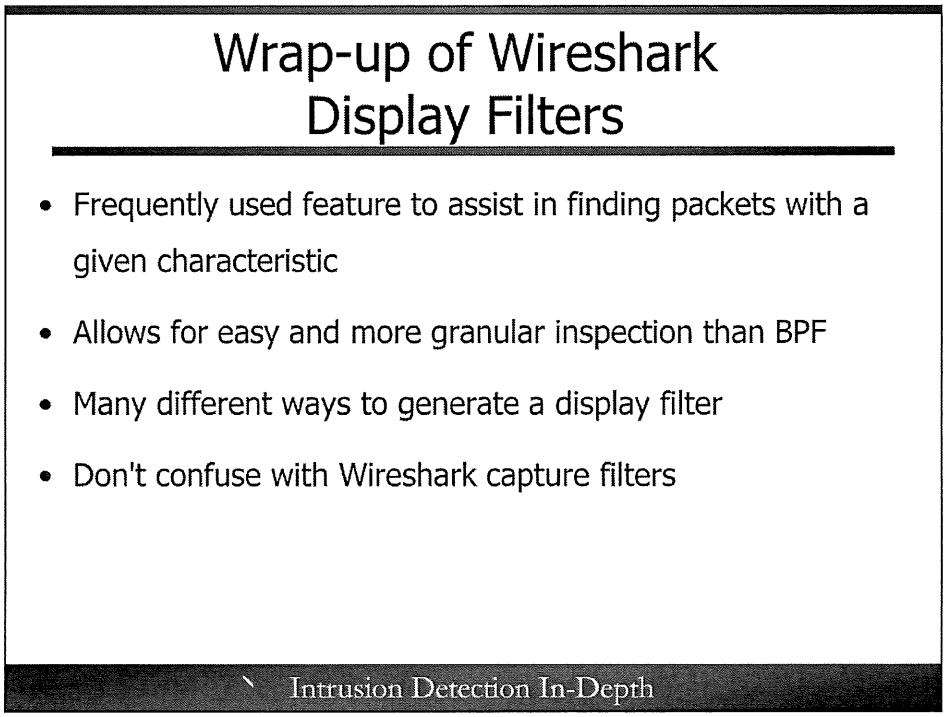
wireshark fragment.pcap



Now save the marked records by selecting File → Export Specified Packets as seen in the upper left display. Finally, save the packets to a file name that you enter and where you save only marked packets as seen in the lower right display. You can now process the new file using any tools that accept libpcap as an input format.

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

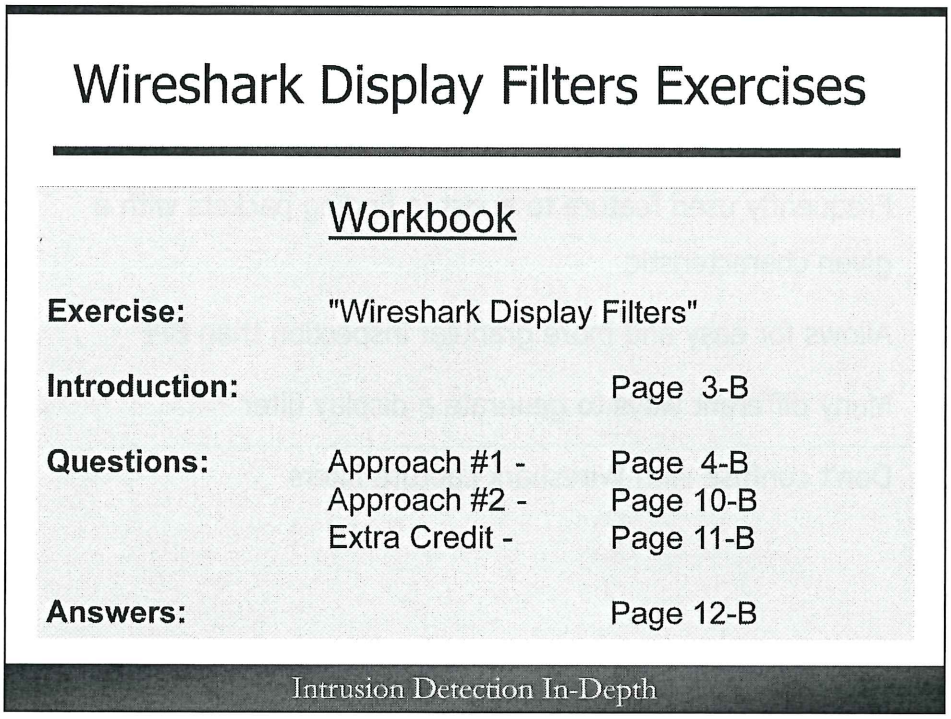
wireshark fragment.pcap



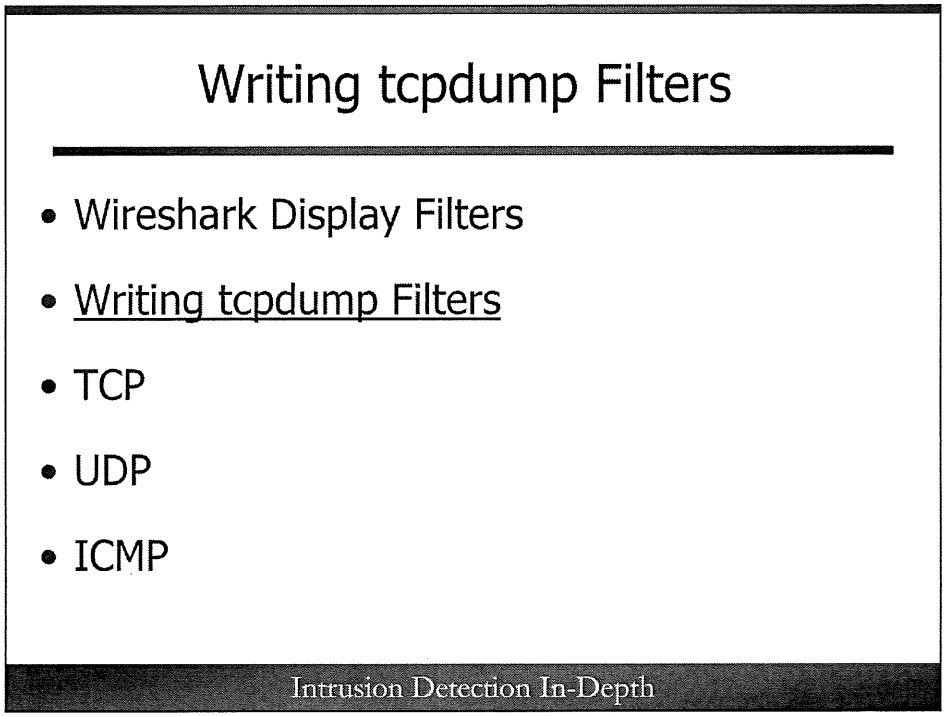
In conclusion, Wireshark display filters are one of the most versatile and often used features of Wireshark. They are extremely adept at finding packets with an attribute that you'd like to examine in one or more packets. As you'll see, they are much simpler to create than BPF expressions and allow you to specify a field of interest more easily. And, Wireshark has more comparison operators than BPF – especially ones that can be used for text searches instead of hexadecimal or decimal.

Wireshark offers many intuitive methods for generating a 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.

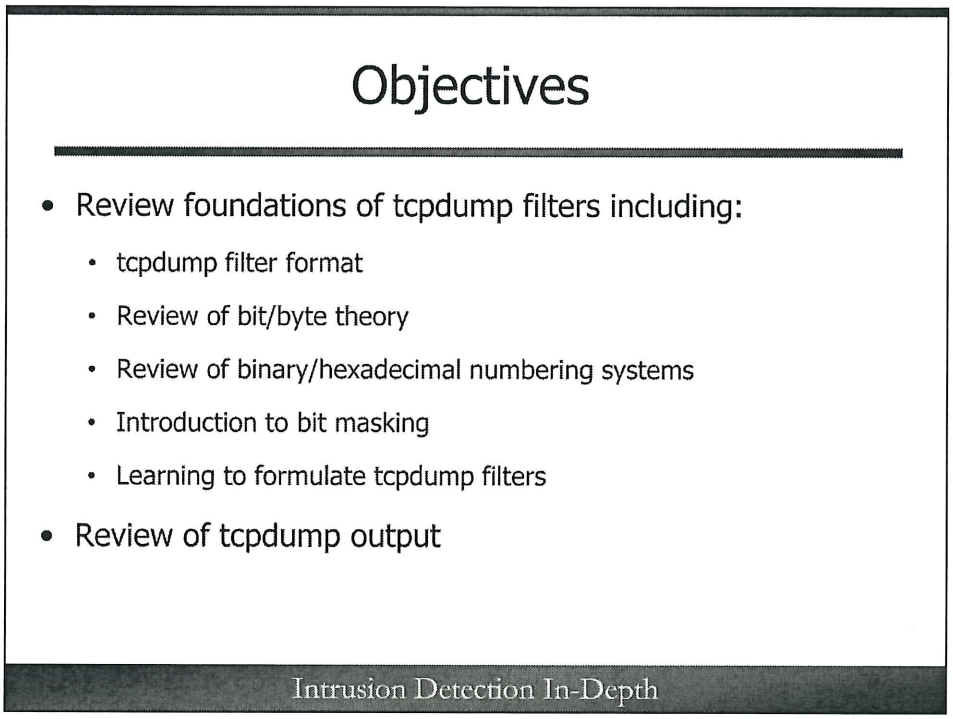
Finally, Wireshark display filters are very different from its BPF capture filters that select packets to be captured when Wireshark is gathering network traffic.



This page intentionally left blank.



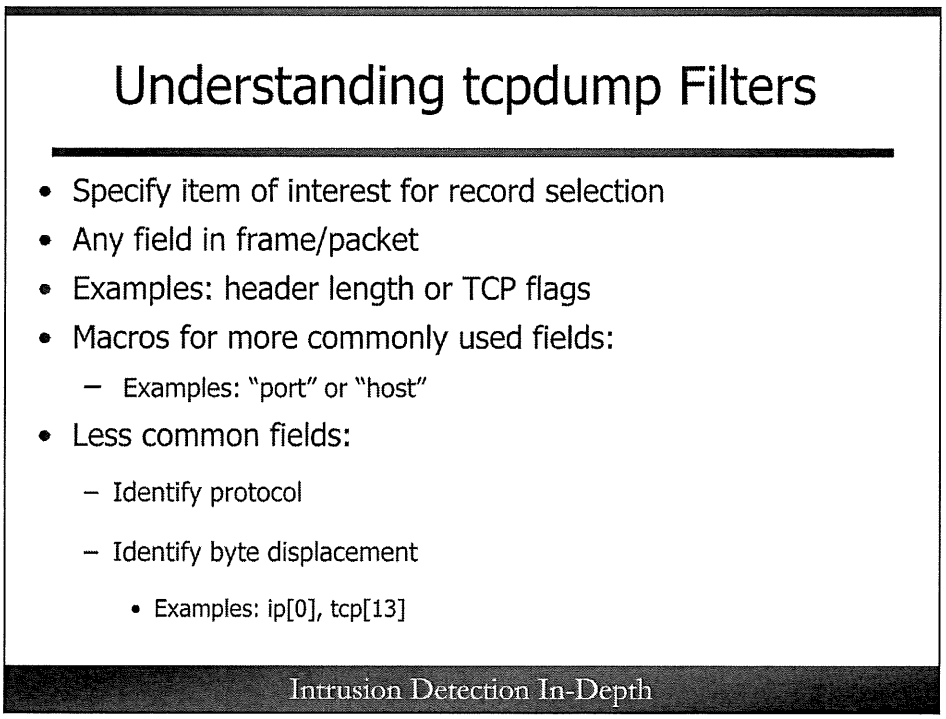
This page intentionally left blank.



Tcpdump filters (also known as BPF – Berkeley Packet Filters) are necessary to selectively gather/read/write records of network traffic. Many times you will want to read packets with specific traits from a file or off the network. This can be done by telling tcpdump what traits and values to process by using tcpdump filters. This is most helpful if you have a large collection of libpcap records that you want to examine for some characteristic. For instance, let's say that you see a SYN scan of a TCP port coming into your network and you've captured the activity using tcpdump. 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 examining records returned from the scanned hosts and ports with both the SYN and ACK flags set. These are the flags that are set when a host listens on a port after receiving a segment with a SYN flag set. These records could be discovered by using a tcpdump filter selecting records with both the SYN and ACK flags set in segments returned from the scanned hosts. Tcpdump filters are very helpful in interpreting network traffic patterns.

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

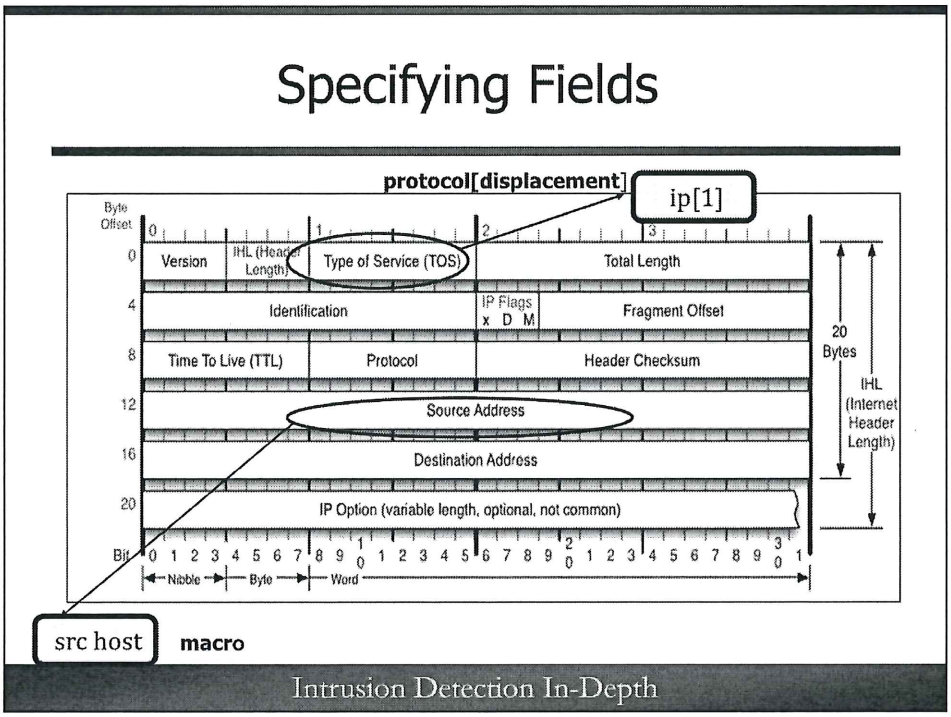
In case you are thinking to yourself that you'll never use tcpdump so you'll never need to understand BPF theory; there are many applications that use BPF besides tcpdump. Snort, Wireshark, and Tshark are few of the many applications that can use BPF. Also, if you ever attempt to do programming with sockets or any of the other applications that can perform network programming, you may find that you 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.



Tcpdump filters need to specify an item of interest such as a bit, a byte, or multiple bytes found anywhere in the packet. Such items 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 provides a designated identification for some types of protocol headers. Much as you would expect, "ip" is used to denote a field in the IPv4 header or data portion of the IPv4 datagram, "tcp" for a field in the TCP header or segment, "udp" for the UDP header or UDP datagram, and "icmp" for the ICMP message. These are not the only protocols that BPF is able to interpret as offsets from the beginning of the protocol header, though these are the ones of primary interest to us. The tcpdump man page has an extensive description of the protocol names available and what you can do with them along with some sample filters.

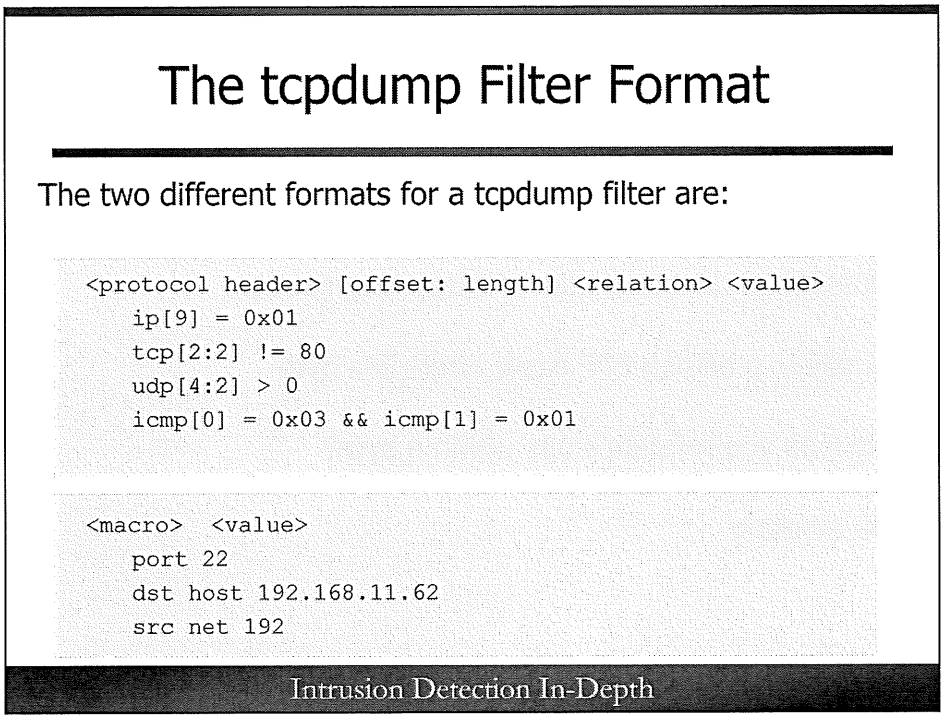
Next, you need to state an offset indicating the starting bye of the field under examination. For instance, ip[0] would indicate the zero byte offset of the IPv4 datagram which happens to be part of the IP header (remember counting starts at 0). tcp[13] would be the 13th byte offset into the TCP header, and icmp[0] would be the zero byte offset of the ICMPv4 message which is the ICMP message type.



Looking at the IP header as an example, we learn two ways to specify different fields. The easier way to specify a field of interest is by using a BPF macro. Not all fields have these macros. The source IP can be specified by combining two macros "src" and "host" to identify the field. But, if we want to look at the type of service field, we have to identify a protocol in which the field is found - "ip" because this is in the IPv4 header, and a displacement in bytes (1) offset in the protocol.

Here are some of the more common macros used in filters. More are discussed in tcpdump's man output.

host	select the record if either the source or destination host matches this IP
net	select the record if either the source or destination subnet matches
	This is useful if there are several IP's from the same subnet of interest to you
port	select the record if either the source or destination port matches
src host	select the record if the source host matches
dst host	select the record if the destination host matches
src net	select the record if the source subnet matches
dst net	select the record if the destination subnet matches
src port	select the record if the source port matches
dst port	select the record if the destination port matches
icmp	select the record if the protocol field ip[9] has a value of 1
tcp	select the record if the protocol field ip[9] has a value of 6
udp	select the record if the protocol field ip[9] has a decimal value of 17



There are two different formats to use for tcpdump BPF. The first format is necessary when there is no macro. First, it requires you to specify the type of protocol where the field you want to examine is found (i.e. ip). Next, you have to specify the byte displacement into the protocol of the field (i.e. [9]). Optionally you can specify the length of the field you are examining, up to 4 bytes[12:4]. Oddly, though , you can filter on a single byte, 2 contiguous bytes, or 4 contiguous bytes, however you cannot specify that you want three contiguous bytes.

If you have a field that is greater than 4 contiguous bytes, you will have to write a separate filter for each 4-byte range. A length designation is necessary if the field length is greater than 1 byte. Otherwise the length defaults to 1 byte without a length value explicitly set. Next, some kind of relation operator is used (i.e. =). Finally, a comparison value must be given (i.e. 1).

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 above 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 with a TCP destination is not 80.

The third filter udp[4:2] > 0 selects any UDP record with a non-zero 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 92.168.11.62.

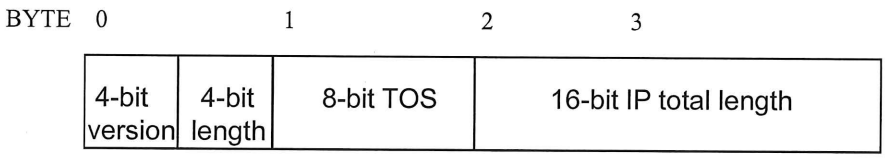
The third macro filter selects any record with a source subnet of 192.x.x.x.

BPF Bit/Byte Fundamentals

- Byte is an 8 bit field
- Possible to denote a span of bytes `udp[0:2]`
- Smallest precision BPF easily offers is a byte
- How do you reference bits within a byte?
- Bit masking

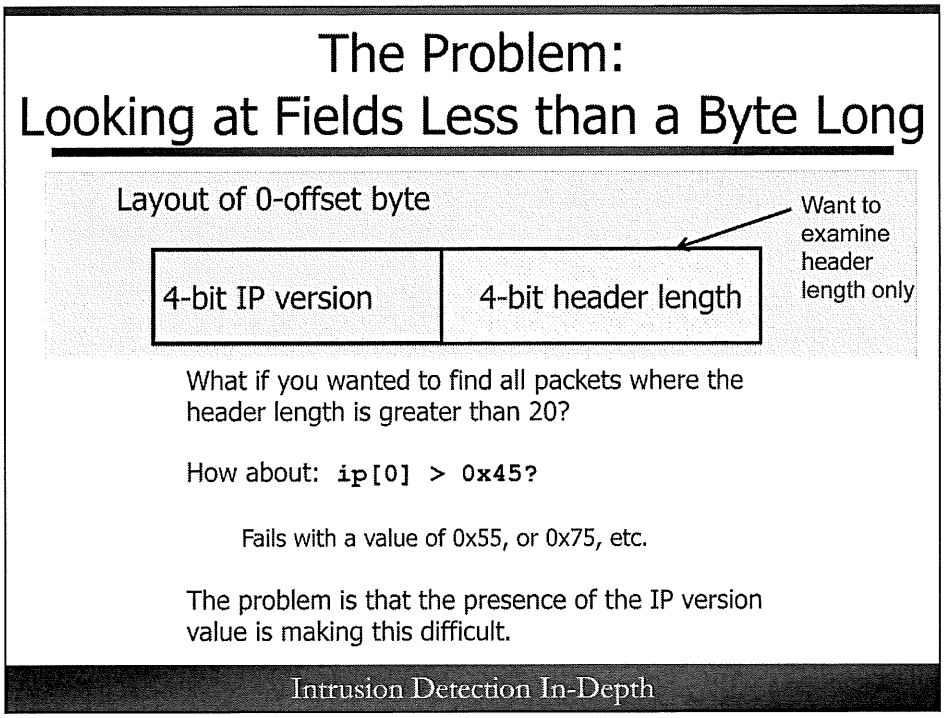
Intrusion Detection In-Depth

First 4 bytes (bytes 0 - 3) of the **IP header:**



This is a refresher of bits and bytes in case you've conveniently forgotten. 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 counting begins at byte 0; all successive bytes fall on these 8-bit boundaries. **udp[0:2]** specifies the byte in the UDP datagram beginning at byte 0 for a length of two bytes.

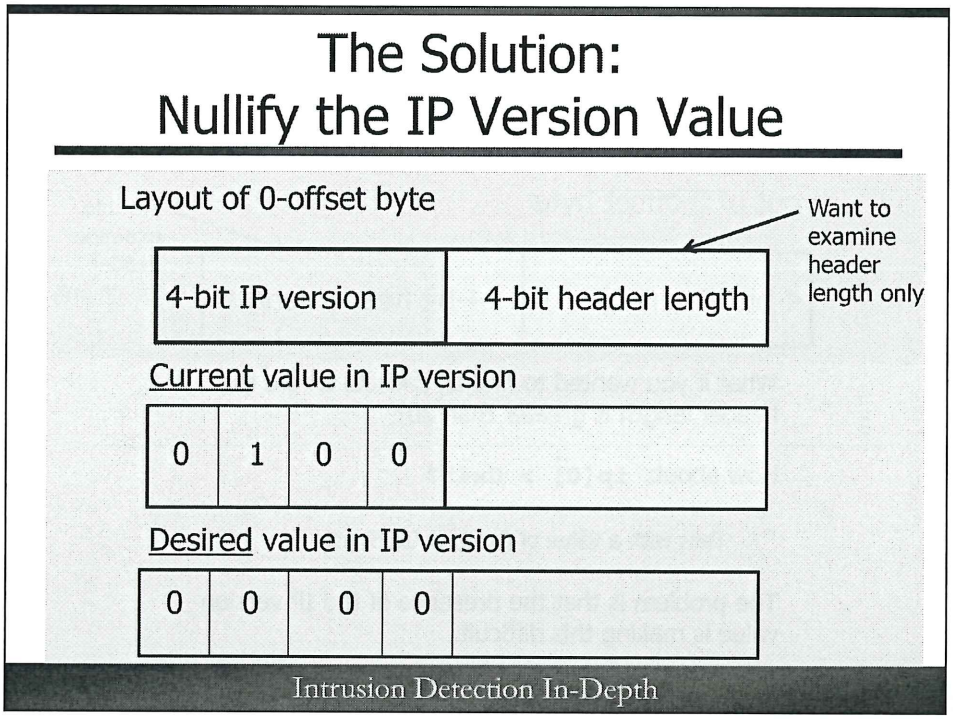
BPF format does not have an easy way to represent a single bit or multiple bits in a byte that you may want to examine. For instance, look at the first 4 bytes of the IP header in the notes section above. If we want to look at the 4-bit version number or 4-bit IP header length, we will have to isolate 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.



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? 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 since it encompasses the 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. Remember that the value in the IP version must be multiplied by 4 to get the true length. That means we want to examine each packet for a value of greater than 5 in the IP header field.

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, not working as we intended. The presence of a value in the IP version complicates our filter. If only we could nullify the value in this field.



As we discovered, there is really no simple operation that is native to BPF that allows us to look exclusively at the IP header length value. But, we can do some operations and manipulations of fields and bits that will allow 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 really are looking 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

- Individual bit or a range of bits may be selected by bit masking
- Uses the Boolean AND operation to keep or discard a bit(s)
- Two bits are AND'ed; the following values yield the following results:

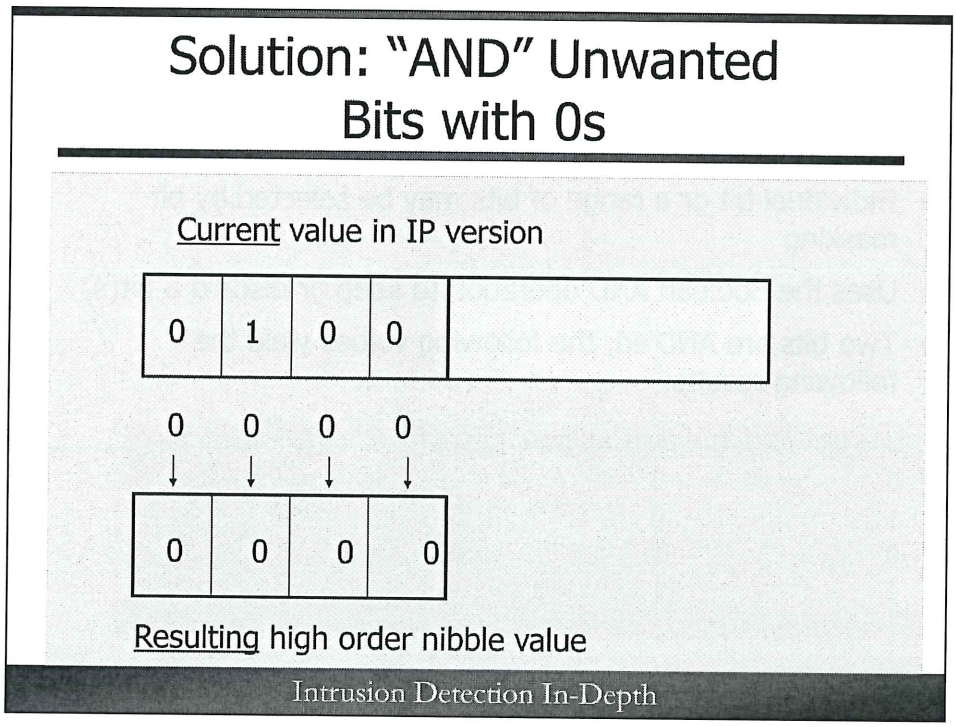
Packet bit	AND	Our mask bit B	=	RESULT
0		0		0
1		0		0
0		1		0
1		1		1

Intrusion Detection In-Depth

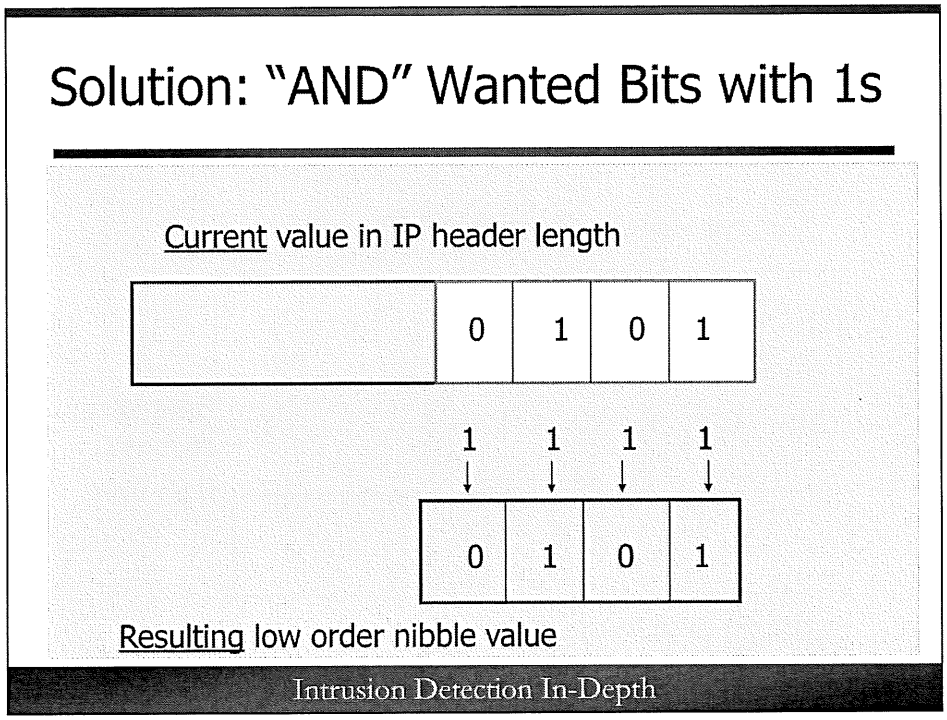
We will use the Boolean AND operation to help us zero out unwanted bits. Let's look at the fundamentals of applying this theory. The AND operator is used because it can easily discard unwanted bits by AND'ing an original bit found in the packet with another bit of 0. This is known as the mask bit; we select the appropriate value – either 0 or 1 to discard or preserve the original bit in the packet. The AND operator can preserve original bit values by using a mask bit of 1.

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

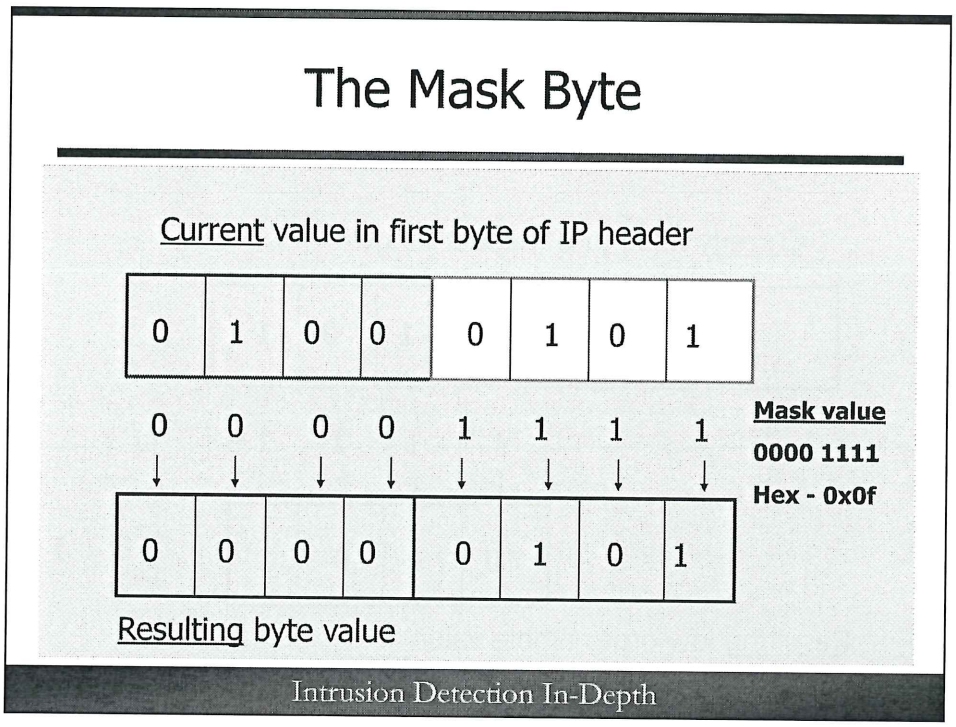
Take a bit found in the original byte, say in the IP header length field, and then select an appropriate mask bit for the AND operation. The appropriate mask bit is one that either preserves the packet bit when the mask bit value is 1, or discards the original packet bit when the mask bit value is 0.



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

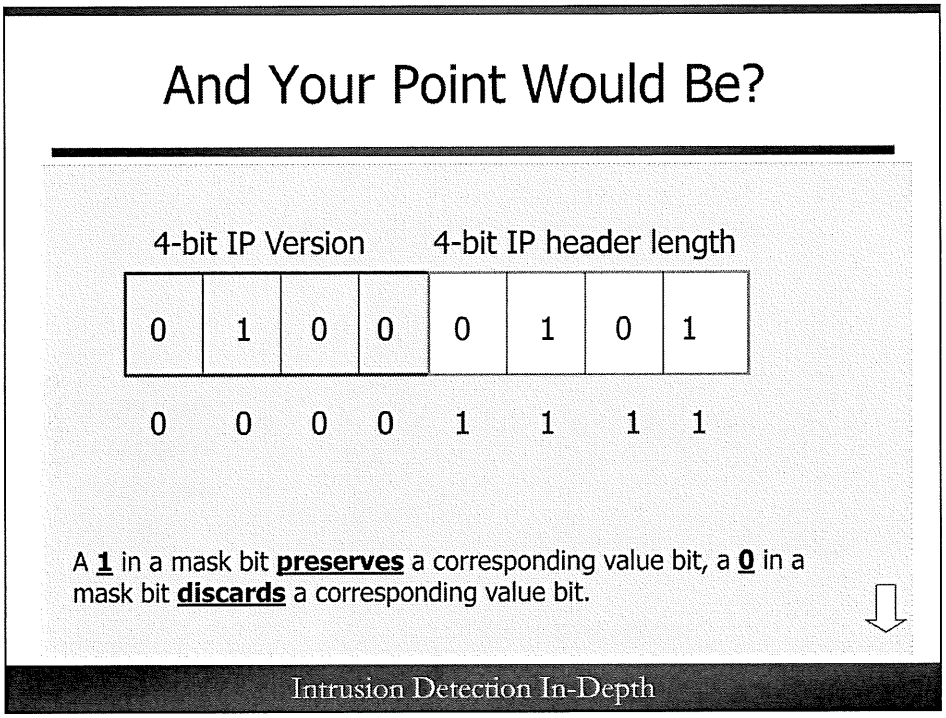


Because we are dealing with an entire byte, we must also pay attention to the low-order nibble, the IP header length that we want to preserve. We must preserve the original value that we found there. We can't simply ignore this field. In order to preserve the current value found in that field, we "AND" all bits with a value of 1. In other words, we have a mask binary value of 1111 for this nibble.



Ultimately, what you have to do is create an appropriate "mask" byte. This is a byte that will be AND'ed with the original value found in the zero byte offset of the IP header to give us the desired resulting byte which will have the high-order nibble of all zeros and the low-order nibble value as it was before the AND operation. So, this just means that our mask byte is a binary 0000 1111 which translated to two hexadecimal characters of 0x0f.

It is important to note that you must specify a prefix of 0x to denote hexadecimal. Otherwise, BPF will think that you mean decimal. BPF will fail on a parse error if you supply a hex value in the mask byte and do not use a prefix of 0x. For instance, if you were to try the following filter 'ip[0] & 0f != 0', BPF does not know that 0f was hexadecimal unless you used the filter 'ip[0] & **0x0f** != 0'. It seems obvious that the value of 0f could only be hex, but BPF forces you to use the 0x prefix for its interpretation.



The easy way to remember how to figure out a mask is to examine which bits need to be preserved as their original values and use a corresponding mask of 1 in the bit. Those bits that need to be discarded should have a corresponding mask of 0.

Now, try your hand at figuring out a mask.

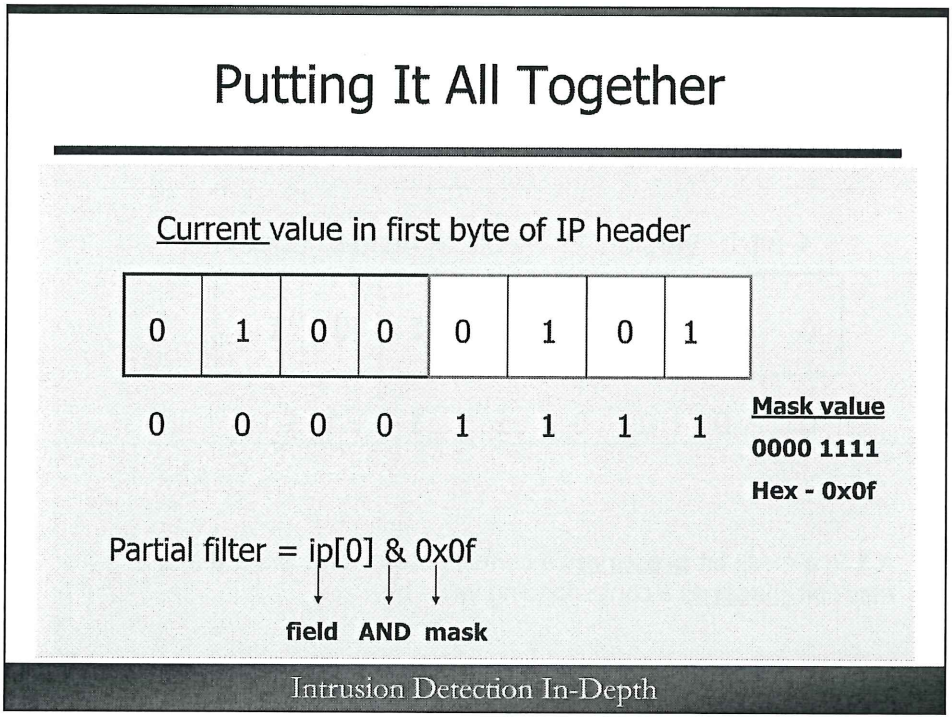
What would the mask be to preserve the high order 4 bits (the version number) and discard the low 4 order bits (the length)? What are the two hex characters that make up the mask byte?

0 1 0 0 0 1 0 1 AND

_ _ _ _ _ _ _ _ MASK? <== **Fill in the blanks with the mask**

0 1 0 0 0 0 0 0 YIELDS

The answer for this can be found at the end of this chapter, Exercise 1.



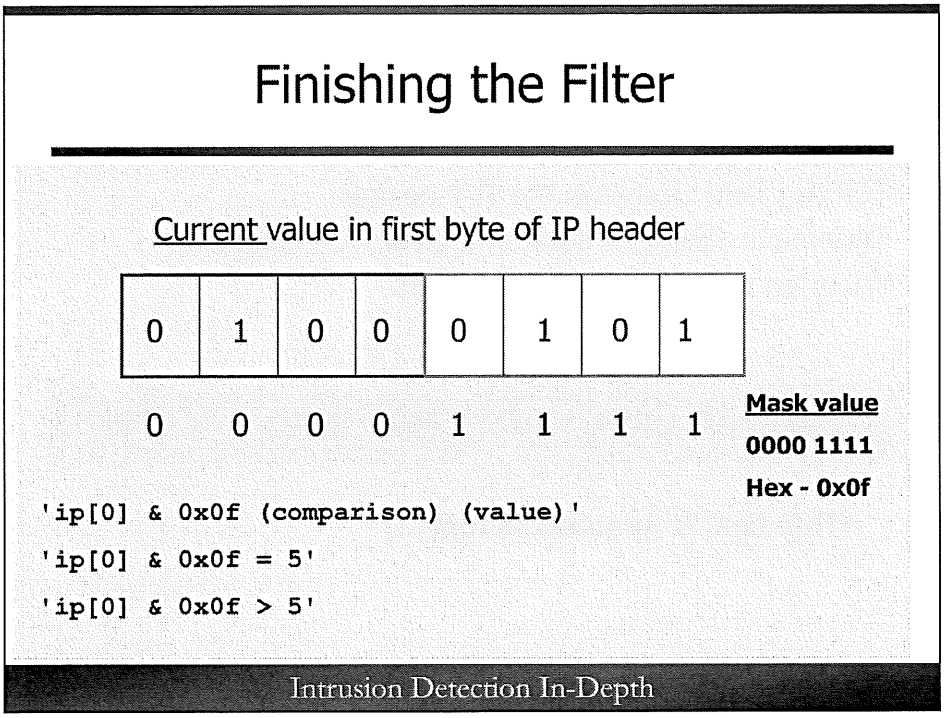
We figured out the mask that we want to AND with the zero byte offset of the IP header, but how do we tell BPF how to do this? What we do is first identify the byte (or bytes) of interest by identifying what protocol we are dealing with (IPv4) and the displacement into the protocol that the byte is found (0 – first byte). Next, we use the "&" symbol to denote the Boolean AND operation and then we must tell it what mask value to AND it with. This is the mask value that we figured out of 0x0f.

Once the mask has been computed to figure out which bits to discard and which to preserve, it has to be "superimposed" over some byte or span of bytes. In this case we need to superimpose the mask over the entire zero byte offset of the IPv4 header because that is where the fields of interest lie. So, in this case that field is represented by **ip[0].** The partial filter of superimposing the appropriate mask over the field of interest becomes **ip[0] & 0x0f**. We will discuss the final piece of the filter in several upcoming slides.

An important point is that we must designate bit masks that are minimally a byte in length. Sure, we 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.

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, while "and" is used to concatenate two different criteria in a single filter, for instance 'tcp port 80 **and** host 192.168.1.2'.

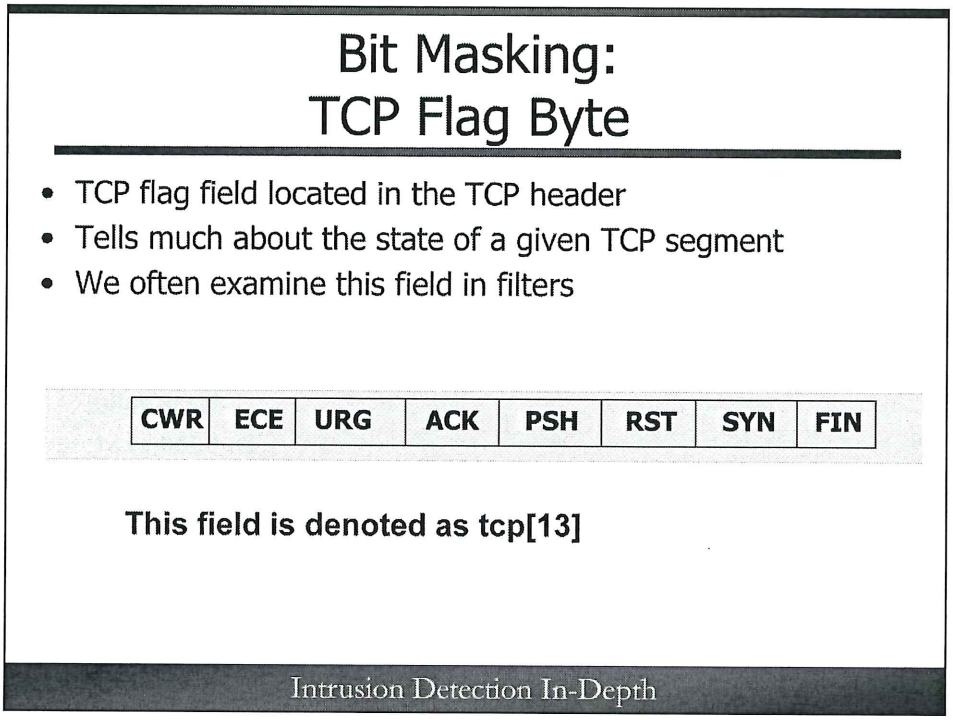
It is important to understand that using a filter of any kind, mask or no mask, does not alter the value in the packets. It simply specifies the criteria used to analyze the packets.



So far you have a byte to match and a mask that selects part of the byte. You now need to specify a comparison operator and a value that performs some kind of assessment. There are several common comparison operators that BPF uses such as "=", ">", "<", and "!=". If you are interested in learning more about the others – look at the tcpdump man page.

The final part of the filter is a value for comparison. The first filter 'ip[0] & 0x0f = 5' examines the IP header length for a value of 5. This finds all IPv4 headers that have no IP options since it is 20 bytes in length. The next one 'ip[0] & 0x0f > 5' finds all IPv4 headers that have IP options.

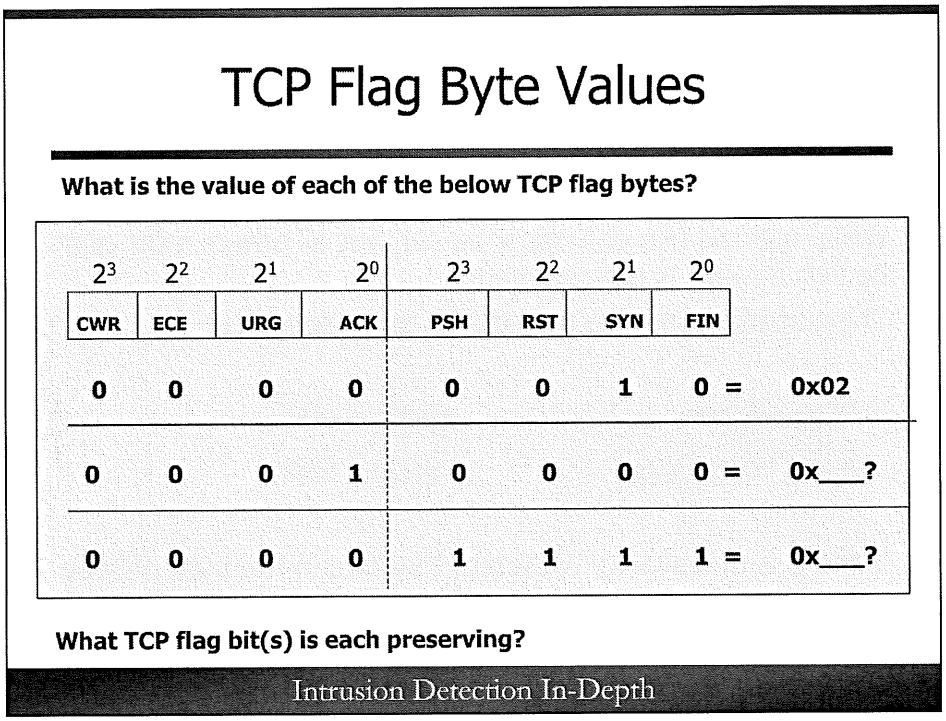
When you supply any kind of filter to tcpdump, make sure that you surround it with single quotes for non-Windows operating systems. Windows uses double quotes. This is to avoid confusion when supplying the filter on a command line. For instance, say you are using a Linux system and give the filter of 'ip[0] & 0x0f > 5', but you accidentally omit the single quotes. Tcpdump will generate a syntax error and the Linux shell in use generates an error "0x0f command not found." Using the single quotes prevents these interpretation issues and confusion with shell command syntax.



There will be many times that 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? Let's say you are interested in viewing only TCP segments with either the SYN or SYN and ACK flags set. This can tell you whether or not a connection was attempted and established. Also, the operating system fingerprinting tool, p0f, looks at the SYN segment for passive OS detection. It can also look at the SYN/ACK segment as well. If you were interested in studying the fields that p0f examines, you could write a filter to select only these segments from traffic.

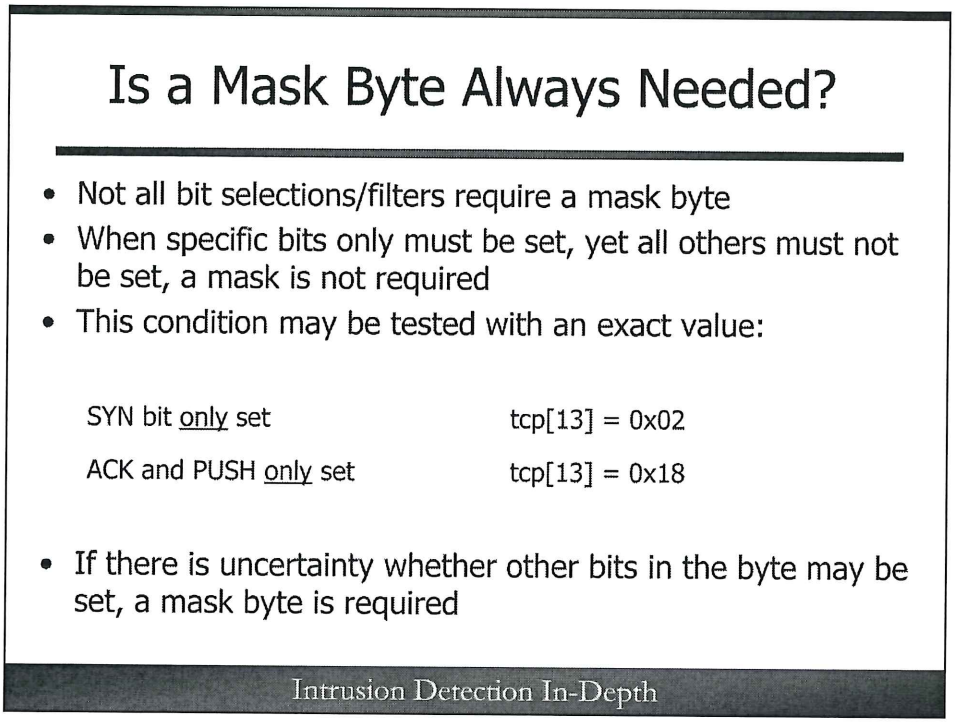
Or, perhaps you are interested in viewing segments with payload only. Typically, you would find these in a segment with the PUSH flag set. Though, it is possible to find payload on segments with the ACK flag only set, or even segments with the SYN or FIN set. The SYN flag is an unusual example; however, some operating systems accept data on SYN. Finding data on FIN packets is not as rare because some web servers often include data on the FIN since this combines the last transmission of data and the close of the session. This is more efficient for busy web servers. We're getting ahead of ourselves since this will be covered in the next section on TCP. But, there are certain concepts you must be aware of now to understand TCP flag selection by bit masking.



In the above slide, we set the foundations for analyzing the TCP flag bits. The first line has the byte value for a TCP flag byte where the SYN flag only is set. The SYN bit falls in the 2^1 position; all other bits are 0. That is how we arrive at a flag byte value of 0x02.

What is the TCP flag byte value for the second and third lines above? What flag bits are selected with those byte values?

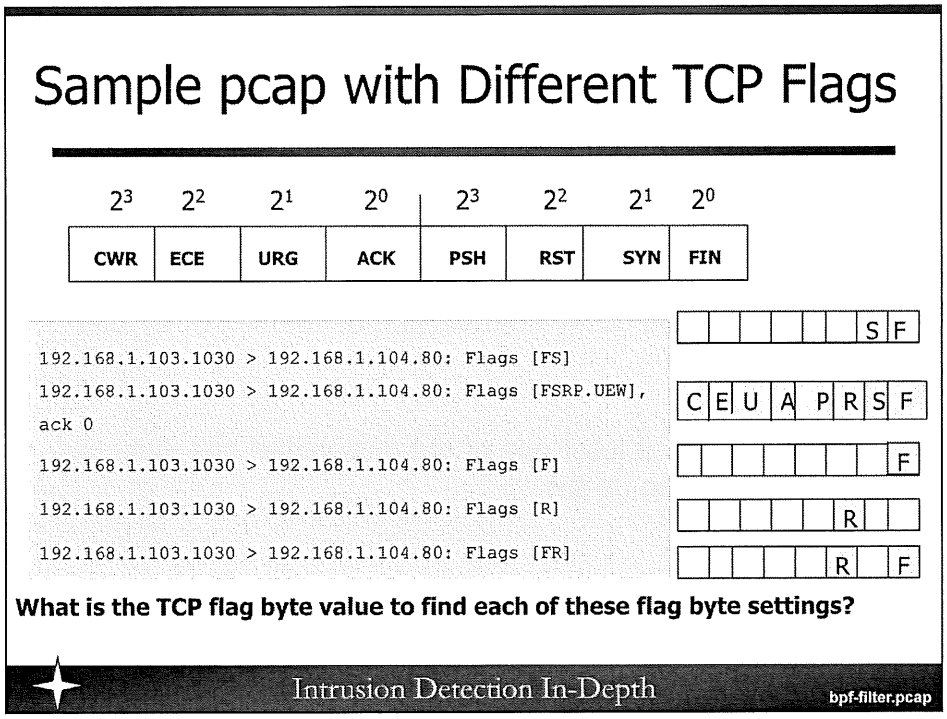
The answers for this slide can be found at the end of the chapter, Exercise 2.



A mask may not be required when dealing with one or more bits in a given byte of interest. If the condition you are designating is an exact value, a mask is not required. This means that 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 therefore you can specify its value as 0x02 or decimal 2, though we'll follow the convention of specifying TCP flag values in hexadecimal throughout this section and course. Perhaps you would like to filter packets where both the ACK and PUSH flag 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 need to discard the high-order nibble and preserve the low-order nibble. The only way to configure the filter is by using a mask byte.

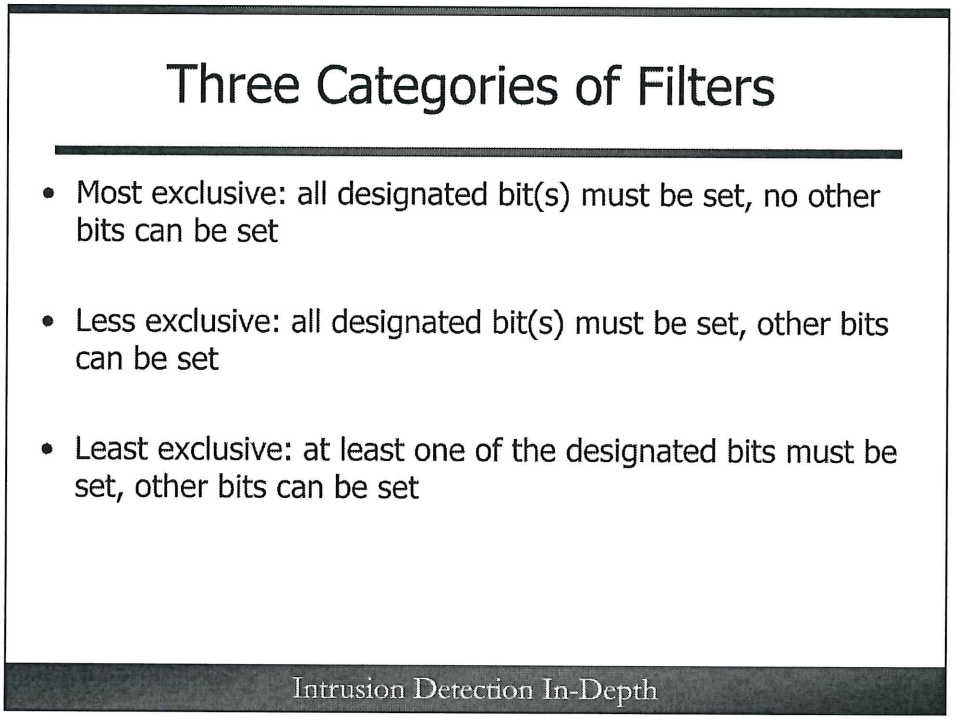


Perhaps the best way to understand different filters for TCP flags is to demonstrate which packets three different, but related, filters select. We'll use the five packets shown above. The flag bits for each packet are filled in to the right of each tcpdump record. For instance, the first packet has the FIN and SYN flag bits set. You see those highlighted in the depiction of the TCP flag byte following the record. Looking at the TCP flag byte depiction on the top of the slide, you see each of the flag bits and the associated bit values.

To find the SYN and FIN bits set:	0x03
To find all flag bits set:	0xff
To find a FIN flag set:	0x01
To find a RST flag set:	0x04
To find the FIN and RST flags set:	0x05

Let's go through finding specific flag bit combinations from a pcap that contains these 5 records.

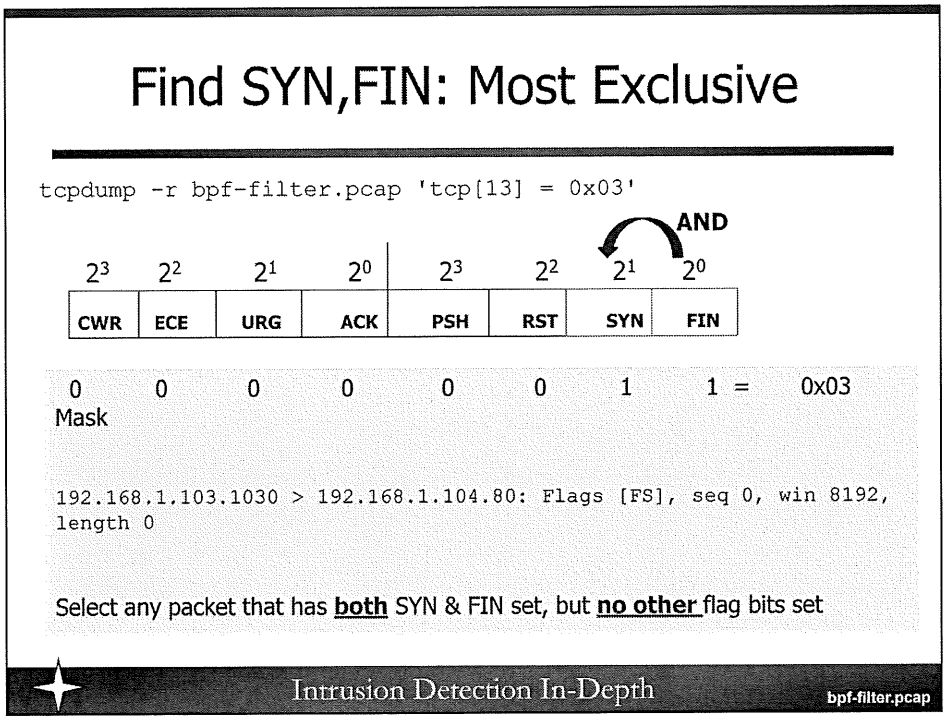
To see the output, enter the following on the command line:
tcpdump -nt -r bpf-filter.pcap



For our purposes, we'll categorize the types of filters we use. The first is known as the most exclusive. This is where all designated bits must be set, yet no other bits 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.

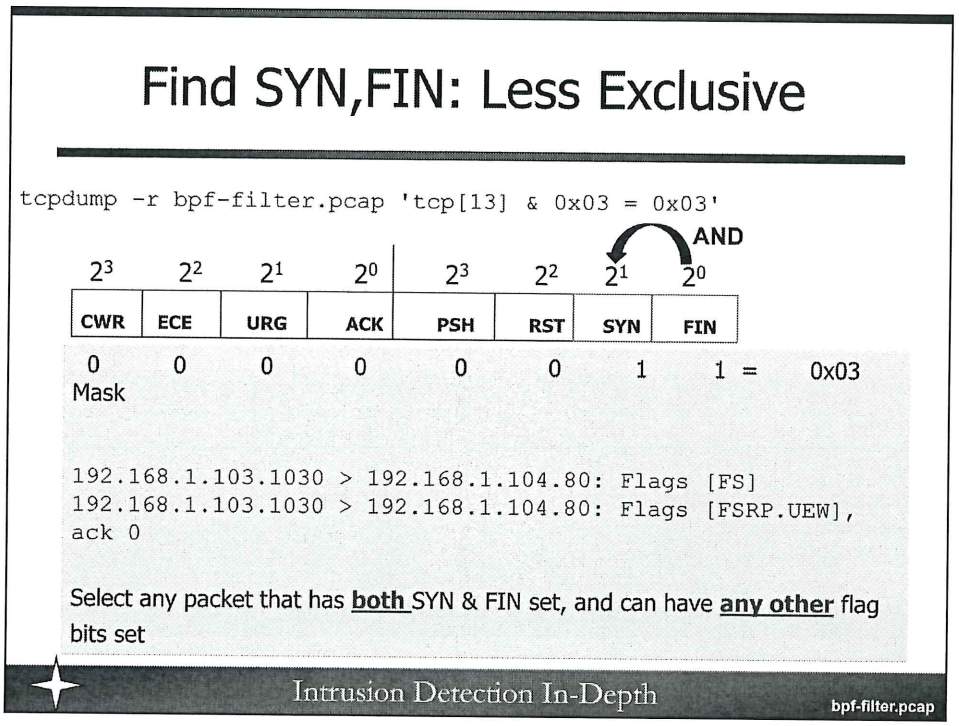
The final category is the least exclusive one. This is where either the SYN or FIN must be set, yet any of the other flag bits may be set too.



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

Let's apply the a 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 tcpdump packet displayed from this filter is the one 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'

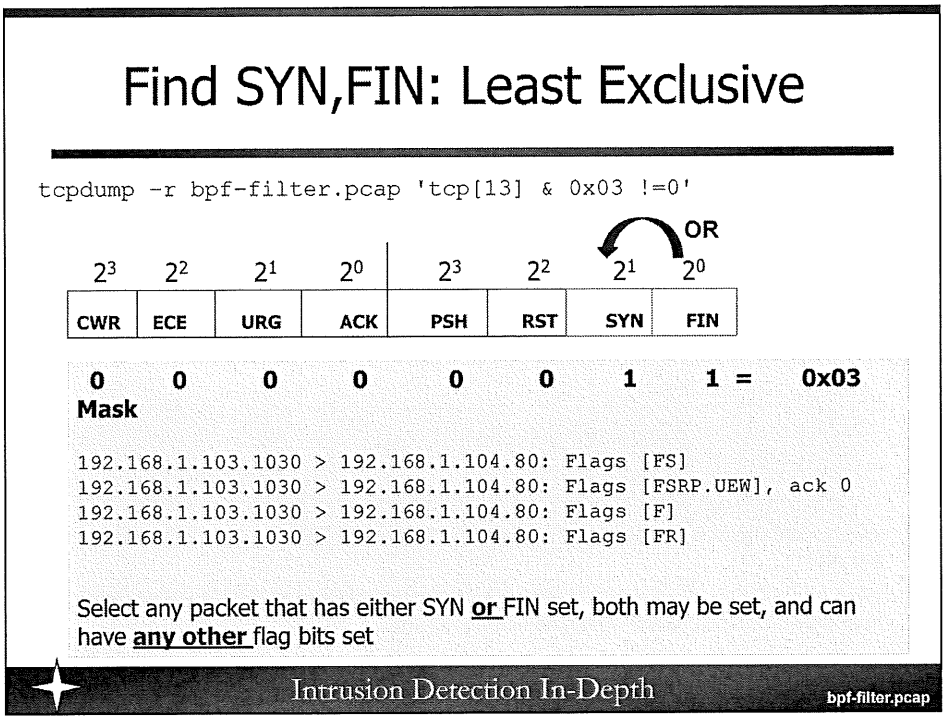


Let's examine a second filter that is less exclusive. Say that we want to find all packets that have the SYN and FIN flags set, but we don't care if any other flag bits are set. We need a way to disregard or discard all other flag bit settings. We need a mask byte in this case since an exact byte value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

The filter 'tcp[13] & 0x03 = 0x03' indicates that we are doing an "AND" operation with a mask byte of 0x03 on the flag byte. The result of this operation must be 0x03. Essentially using a mask of 0x03 causes every other bit in the flag byte to be 0 by AND'ing it with 0. The resulting value is necessarily 0. This essentially means that we disregard all value/bit settings of every other flag field except those of the 2^0 and 2^1 bits. We are saying that we do not care what value is in any of the other bits.

The result must yield 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'



Finally, let's ease up even more using the 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 in the flag byte to be masked with a 0. This means we disregard the value of any of the other flag bits.

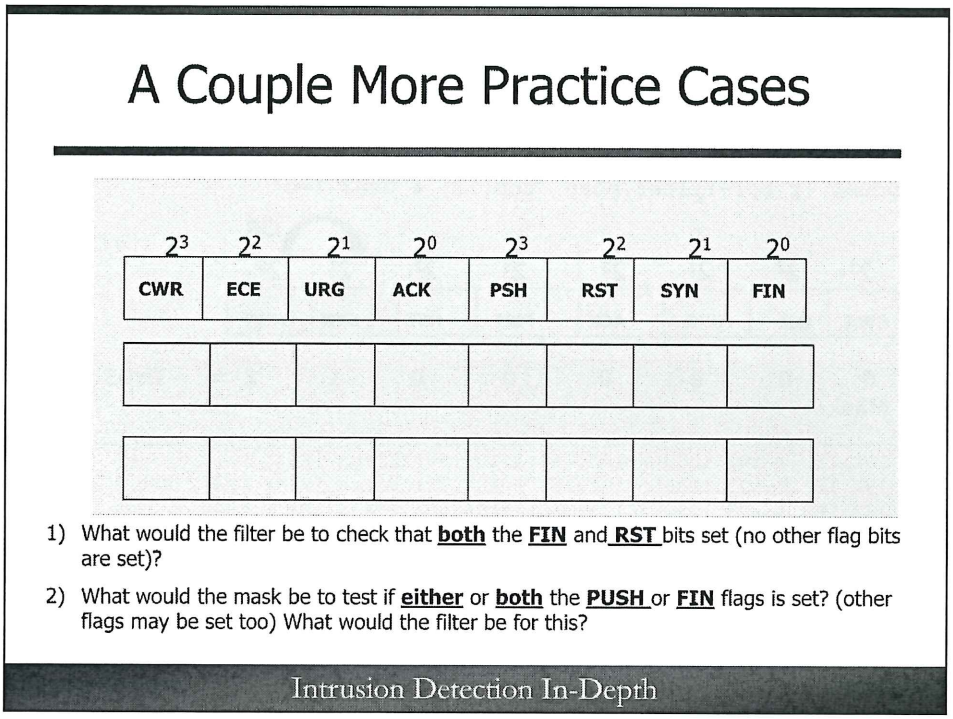
If the SYN and FIN flags are both set the result of the mask operation is a value of 0x03, and is non-zero. This also means that if the SYN flag alone is set, the result of the mask operation is a value of 0x02, and is non-zero. Finally, this also means that if the FIN flag alone is set, the result of the mask operation is a value of 0x01, and is non-zero. As you see by the tcpdump output, the filter has selected all packets that have either or both the SYN, FIN flags and may have other flags set too.

Perhaps you are thinking that since this logic is confusing – why not test for either the SYN bit set or the FIN bit set – something like tcp[13] & 0x01 = 0x01 or tcp[13] & 0x02 = 0x02. This may be manageable for this example, but suppose you want to test a lot of different bit settings; this could be tedious and cumbersome.

Then why is the syntax "!=0"? Think of it this way - what if we used a mask of 0x03, yet only the RST bit was set? What would the outcome be? The result would equal 0 since neither the SYN nor FIN is set. Now, what if the ACK and URG were set? Using the same mask, the outcome would be 0 again. You see where we're going with this – any time that neither the SYN nor FIN is set the result is 0. That's why the "!=0" works succinctly without doing a whole bunch of different masks and tests. Make sense? The whole notion of bit masking and filters can be complex and daunting, but with some practice examples like those found in the hands-on exercises, the concepts will become clearer.

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

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



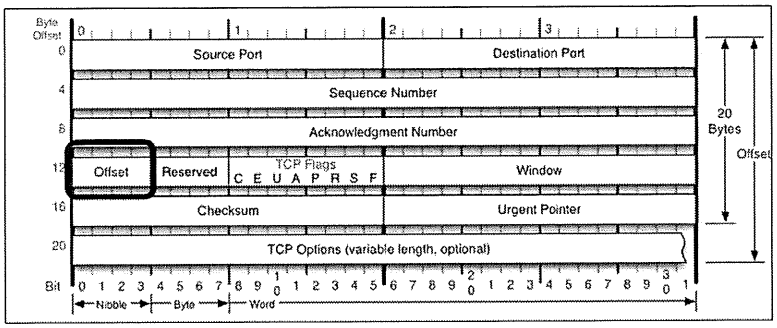
Here is a worksheet for figuring out the above filters:

Filter 1 = _____ [____] = 0x____ ____
protocol [bytes]

Filter 2 = _____ [____] & 0x____ ____ != 0
protocol [bytes]

The answers for this slide can be found at the end of the chapter, Exercise 3.

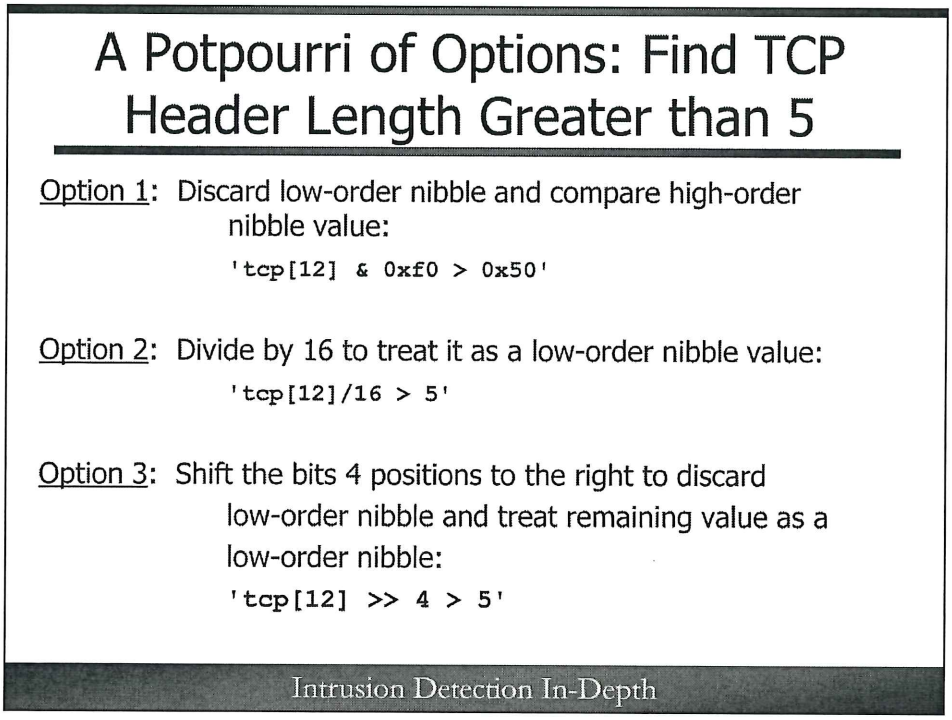
Extract TCP Header Length



TCP header length is found in the high-order of 12th byte offset of TCP header

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 header length. A standard TCP header with no TCP options is 20 bytes long. 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. Now, remember that the value is found in the high-order nibble of byte 12, meaning that we need to be aware of the value in the entire byte, not just the nibble. For instance, say that there is a value of 5 in the TCP header length and we manage to discard the low-order bits. The value in the byte is actually 80 since the bits that are set fall in the positions of $2^6 + 2^4$ or 64 + 16 – or if you care to express it in powers of 16, it is $5*16^1$. On the next slide we will examine several different ways to deal with this particular field.



There are several ways to deal with the issue of the value being in the high-order nibble. But before we investigate your options, you need to make sure that whenever you have a value in the high-order nibble such as we do, remember that you need to manipulate that value. In other words, you cannot just compare the value to examine if it is 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 next page given the deluge of new information you are assimilating.

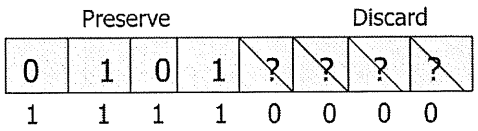
The first option is to leave the high-order nibble value intact, use bit masking to nullify the low-order nibble, and compare the remaining value with 0x50, keeping the comparison value expressed in the high-order nibble. Alternatively, we can divide by 16 since the high-order nibble houses a 16^1 value that is 16 times greater than the 16^0 value found in the low-order nibble.

A third option is to send the low-order nibble into "oblivion" by shifting 4 bits to the right. Don't worry if all this seems confusing just now. Let's take a closer look at each of these options in the next three slides.

Option 1: Mask and Alter Comparison Value

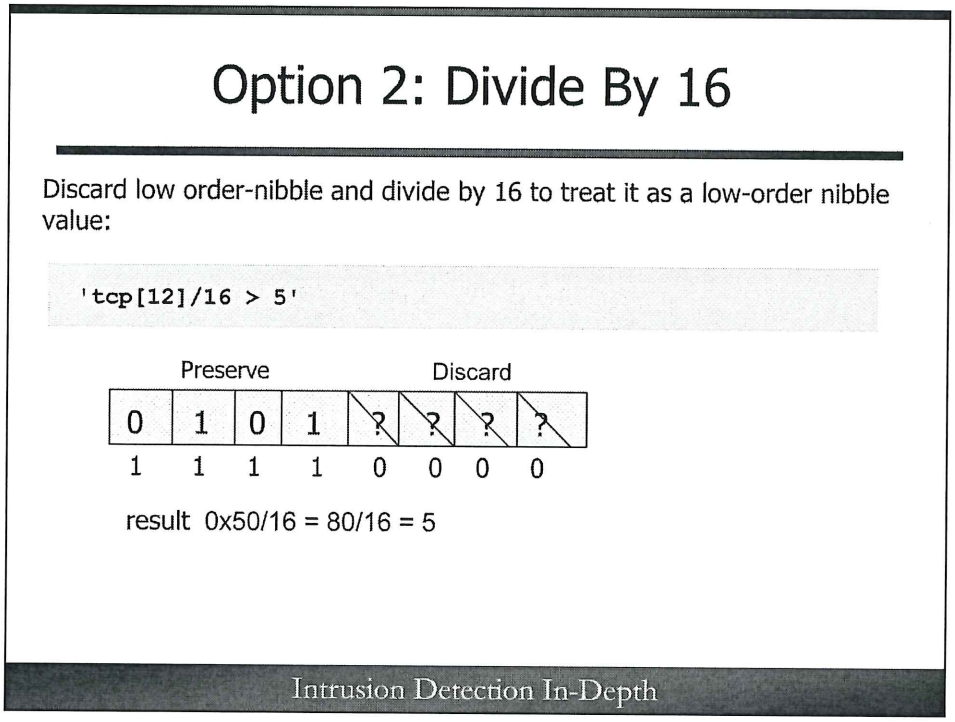
Discard low-order nibble and compare with high-order nibble value intact:

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



result 0x50

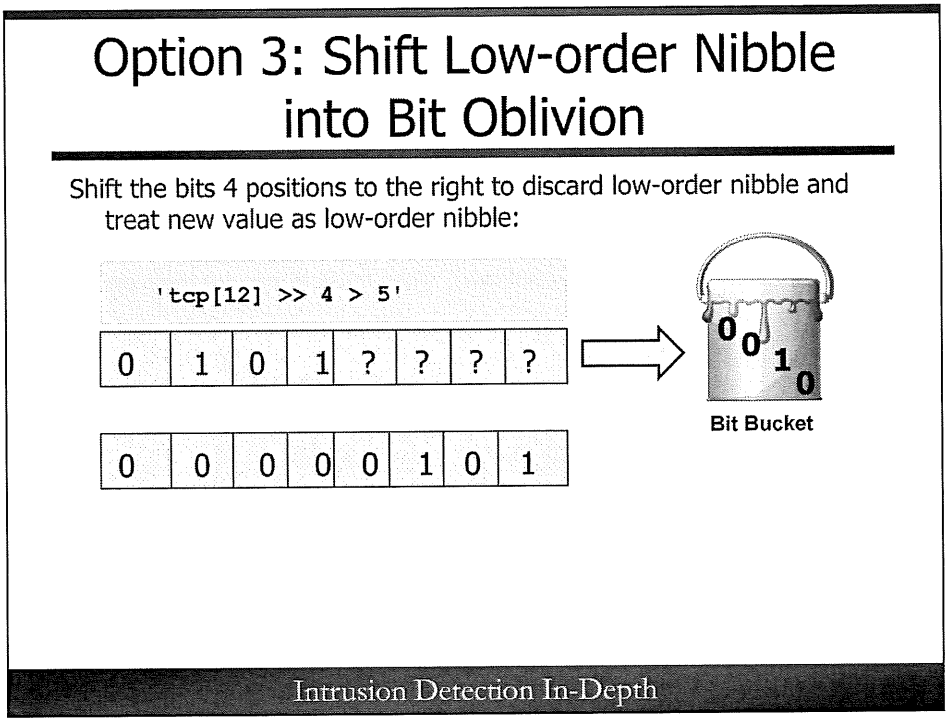
Option 1 is nothing new in regard to discarding unwanted bits. It is what we've been doing all along; however, now our attention is on the high-order nibble. We nullify the low-order nibble by using a bit mask of 0xf0 and performing a binary AND operation on it. We are left with the high-order nibble value intact. We can choose to simply compare it against a value represented in the high-order nibble – in this case 0x50. If you are more comfortable with decimal, you multiply the comparison value of 5 by 16 to arrive at decimal 80 in lieu of 0x50.



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-nibble falls in the 16^1 position where 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 to essentially make 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. Finally, it also supports negation.

One thing to remember about BPF is that it is a low level language that does primitive, but necessary operations. Yes, compared to the facility of all the many ways to create Wireshark filters, it is downright crude. However, since it is at such a low level, bare metal – if you will, language it is far more efficient than all of the abstraction and processing required by Wireshark filters.



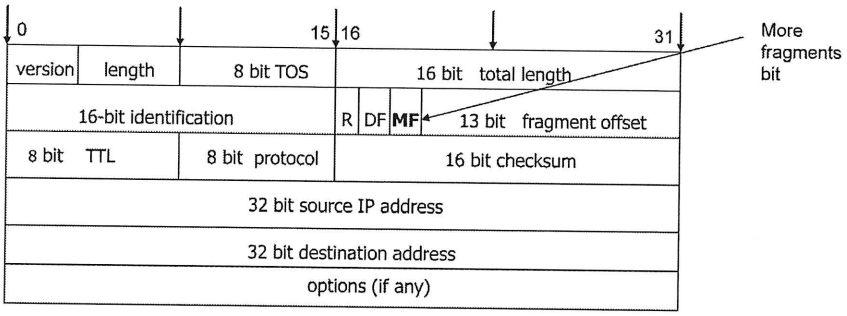
Finally, the easiest way to do this is to discard the low-order nibble value by shifting it right 4 bits. A shift operation moves the bits right or left. The ">>" denotes move to the right and "<<" to the left. You need to designate the number of bits to shift. In this case, we simply shift the values 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.

Essentially, 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 actually divided by 16 in a different way.

One Final Mask Test (Hallelujah!)

Examine the IPv4 header to see if the more fragments bit is non-zero. What is the filter for a non-zero MF value?

Begin by figuring out how many bytes into the IPv4 field this is (remember counting starts at 0 and there are 4 bytes in each row). Then mask all bits but the more fragments bit.



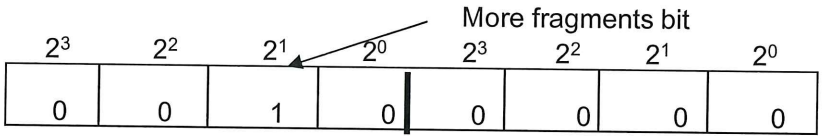
Intrusion Detection In-Depth

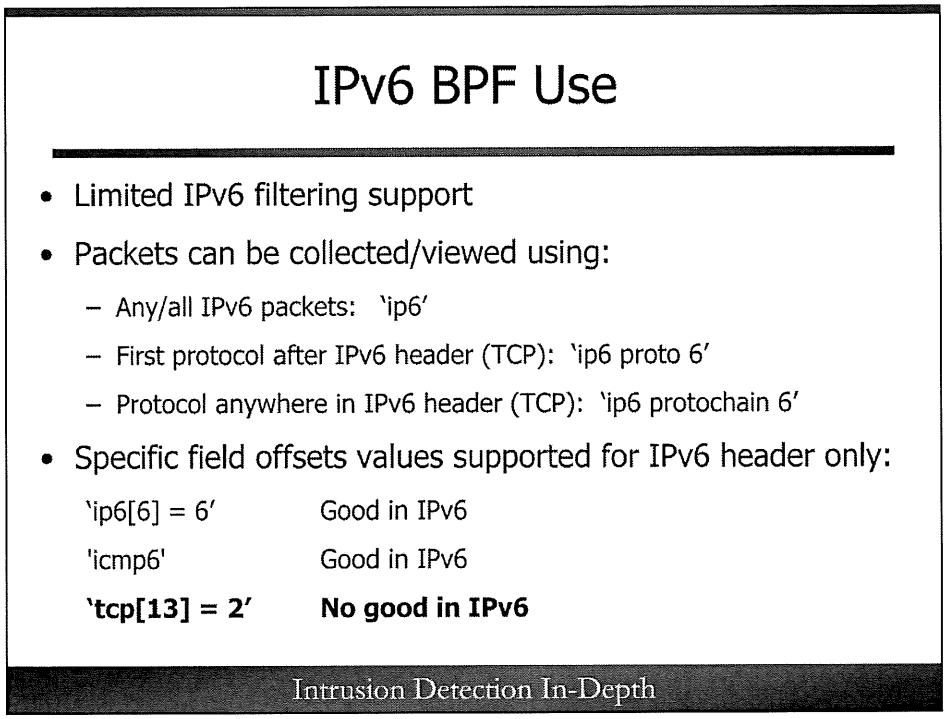
First, let's figure out the displacement of the more fragments bit into the IPv4 header. There are 32 bits per line depicted in each IPv4 header row above. This means that there are 4 bytes in each row. Starting at byte 0, the first row goes from bytes 0 - 3. The second row goes from bytes 4 - 7. We see that the more fragments bit is in the 6th byte.

It is the 3rd bit from the left in the 6th byte. Since we are not at all concerned with the low order 4 bits, we know that the 2nd hex character in the mask can be 0. Now, we are only dealing with the high-order 4 bits. We are only concerned with the 2nd bit from the right of the high-order 4 bits. If that is set, what would the value be? It would be 2.

Remember any of the other bits in the byte can be set.

Look at Exercise 4 answers at the end of this section.





BPF filter support in IPv6 (as of version 4.0.0) is either incomplete assuming they intend to support it or complete and disappointingly inadequate. The support for viewing or collecting packets based on fields in the IPv6 header is fine, but examining packets based on fields/values in embedded protocols is not supported.

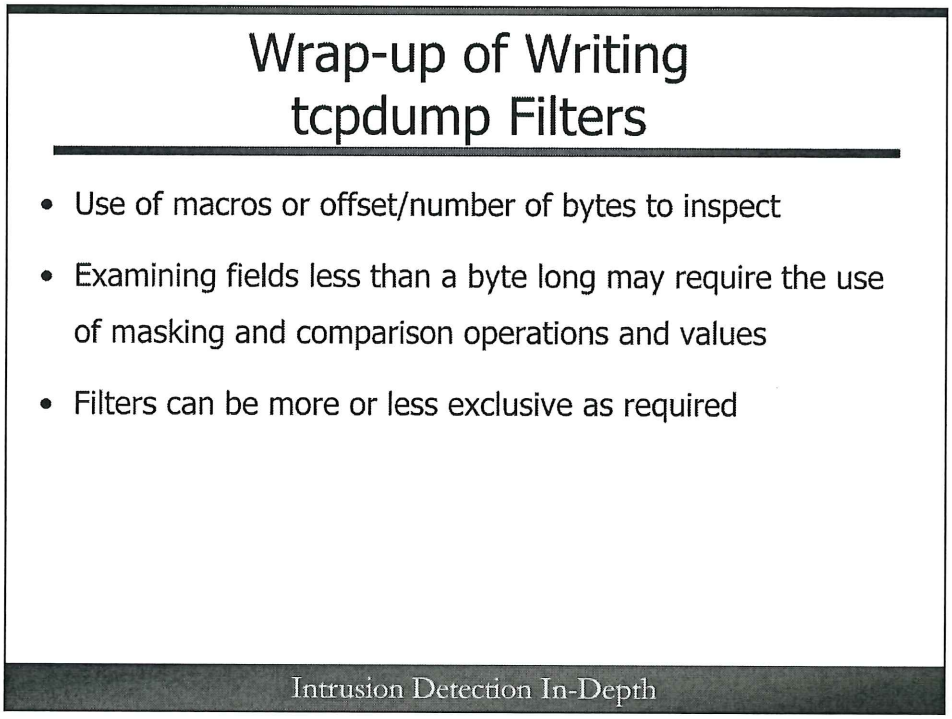
To view IPv6 traffic, you specifically need to give a filter of 'ip6'; otherwise, default tcpdump support will view or collect IPv4 traffic. You can filter per protocol following the IPv6 header by using either the 'proto value/name' or 'protochain value/name' format. The former looks for the specified protocol to follow immediately after the IPv6 header. The latter indicates that the specified protocol may be found anywhere after the IPv6 header. For instance, the expression 'ip6 proto 6' indicates that protocol 6 (TCP) must be the next header value found in the IPv6 header. 'ip6 protochain 6' means that TCP can be the next header value found in any header or extension header following the IPv6 header. As an example, it could follow a fragment header or a Hop by Hop header, etc.

Also, using a filter of 'icmp' does not work on IPv6 traffic since it is looking for ICMP version 4; 'icmp6' should be used instead. ICMPv6 has a protocol number of 58 while ICMP version 4 has a protocol number of 1.

The previous filters that we've created for IPv4 that use a byte displacement and value apply now only to the header portion of the IPv6 packet. They can no longer be used with any of the next headers like TCP. The tcpdump man page says the following:

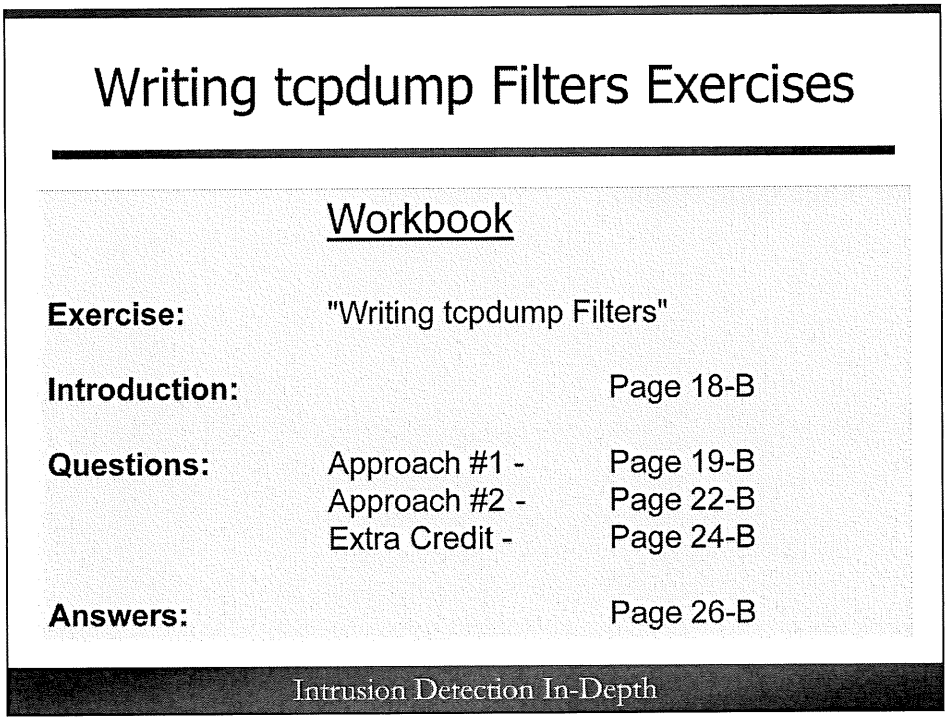
"Arithmetic expression against transport layer headers, like tcp[0], does not work against IPv6 packets. It only looks at IPv4 packets."

This makes it difficult, if not impossible, to examine embedded protocol values. Thankfully, Wireshark and tshark have no problem with this.

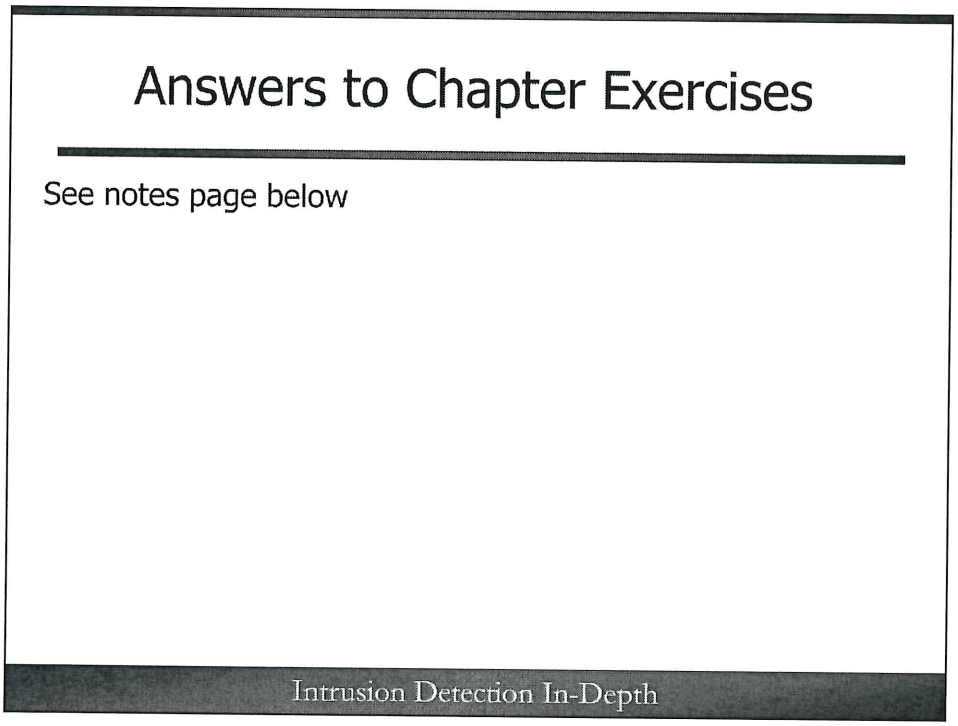


In conclusion, the use of BPF is very straightforward when you are examining a field or fields that fall on byte boundaries. In fact, there are even some macros that can be used to identify commonly used fields. If no macro is available, you reference a field by its protocol and displacement in bytes 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 messier when we examine fields that do not fall on byte boundaries. We may have to use bit masking to selectively preserve and discard original bit values to yield the desired 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. We 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 non-zero value. This permits us to designate a choice of bit settings/values.



This page intentionally left blank.



Exercise 1:

The mask value should be 1111 0000 which in hex is 0xf0. The filter would be ip[0] & 0xf0.

Exercise 2:

The second line would select the record if the ACK bit is set:

The byte value is: **0x10**

The third line would select a record if any/all the PSH, RST, SYN, FIN bits are set:

The byte value is: **0x0f**

Exercise 3:

The filter for both the RST and FIN bits set would be:

tcp[13] = 0x05

The mask for either the PUSH or FIN bits is **0x09**

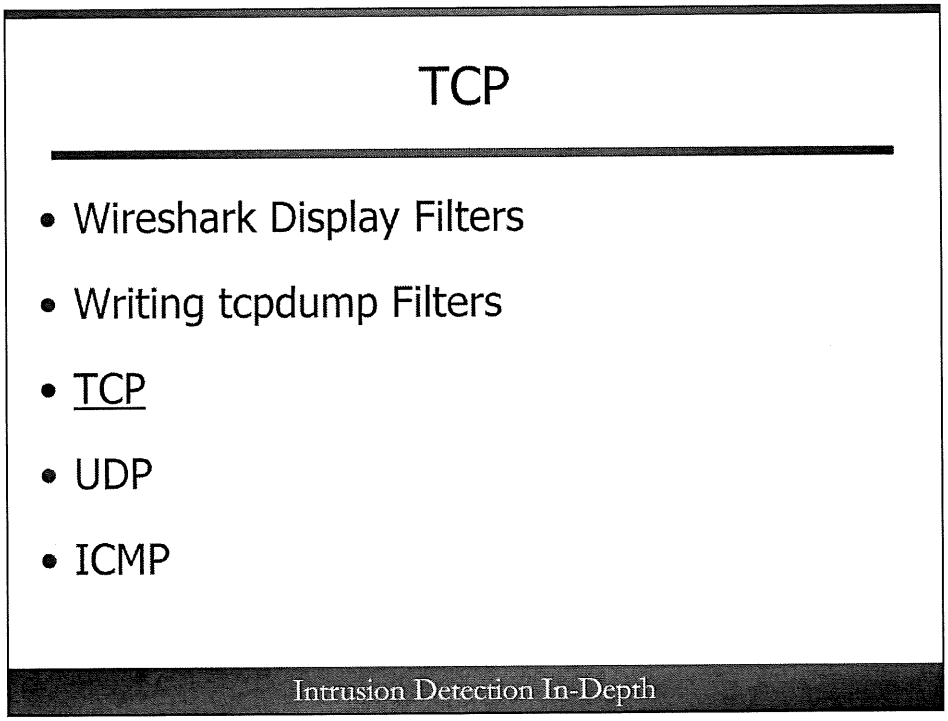
The filter is:

tcp[13] & 0x09 != 0

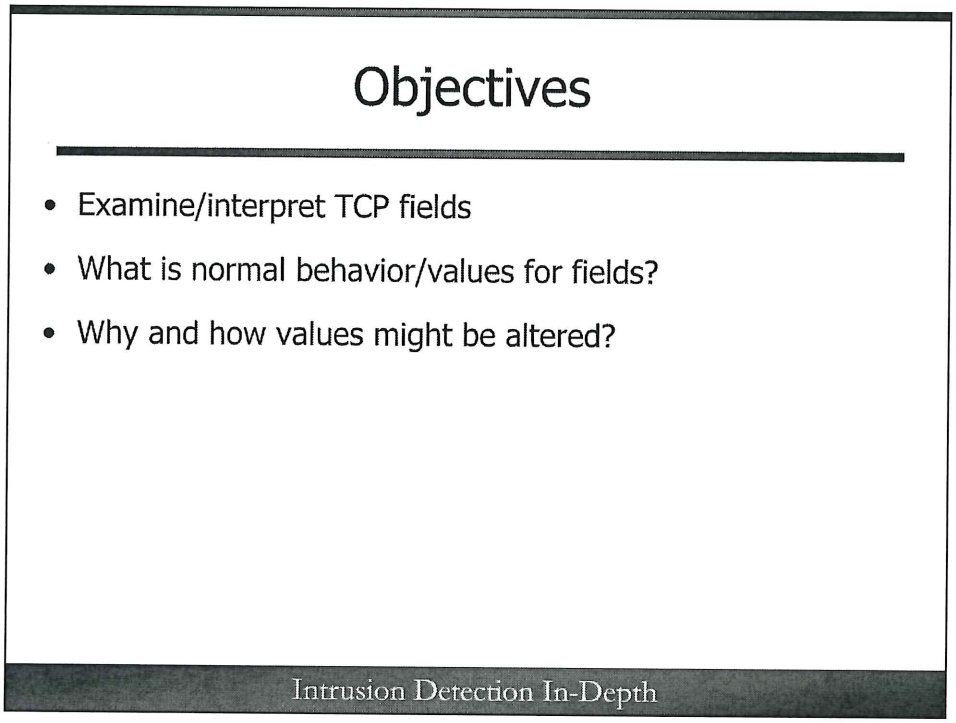
Exercise 4:

The mask would be 0x20. The filter would be:

ip[6] & 0x20 ! = 0



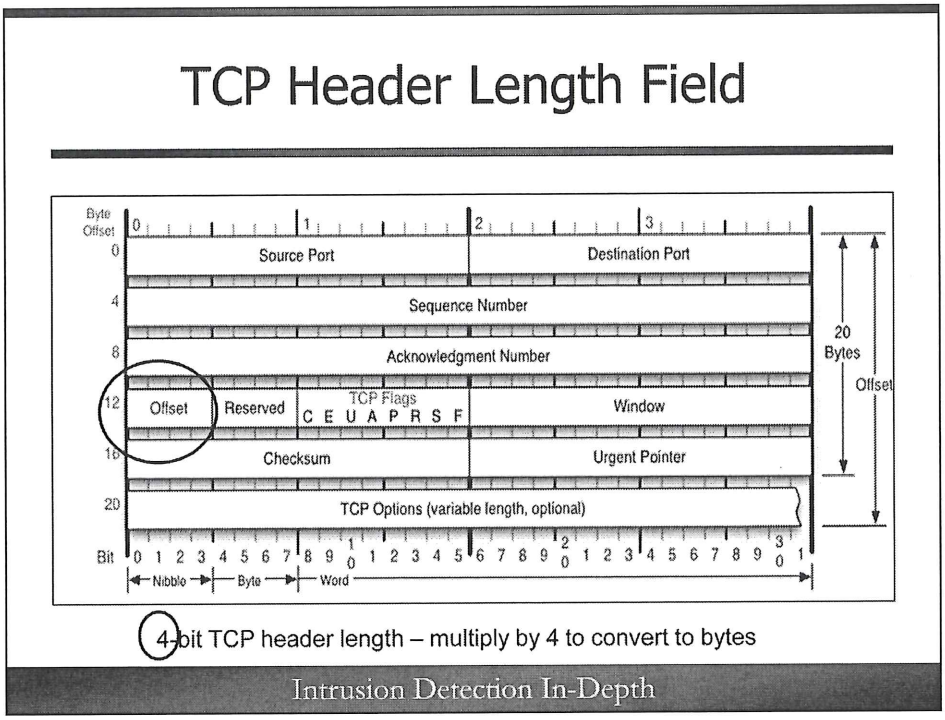
This page intentionally left blank.



We've finally arrived at the transport layer. The first of three protocols that we'll examine is TCP. TCP is a complex protocol with many concerns for us as analysts. It's important to understand the 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 or not packets have been received.

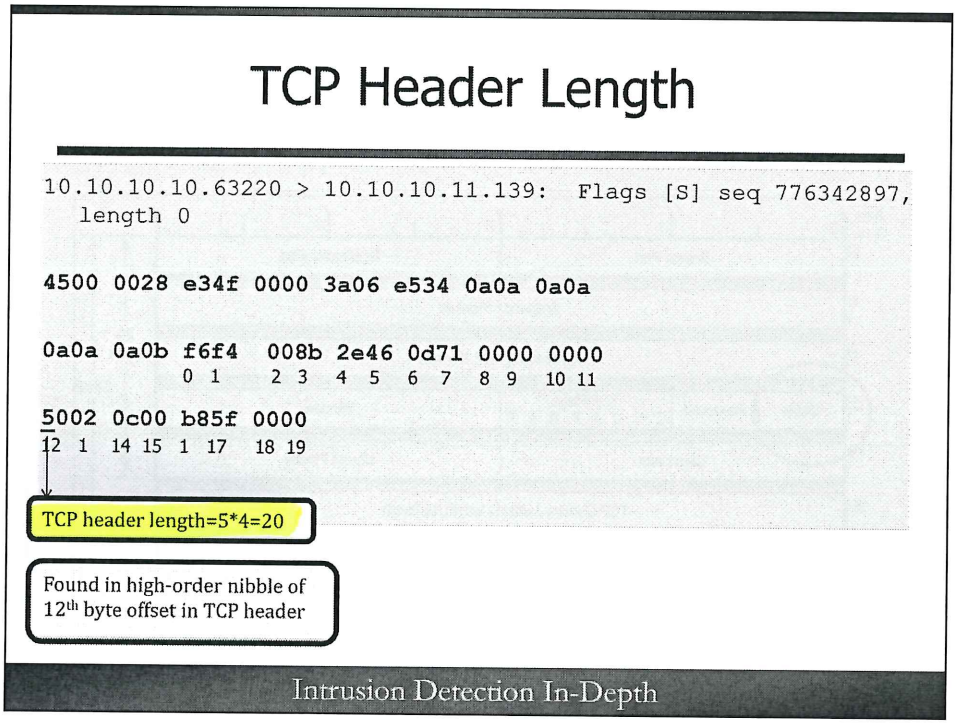
Also, TCP has a true notion of a session. One of the characteristics of a session is that there are multiple individual packets containing TCP segments (term used for a TCP header and data) that comprise a single session. These segments are associated with each other and must have a way to indicate chronology.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

We will become familiar with all these concepts and more to understand normal TCP behavior and potential deviant behavior.

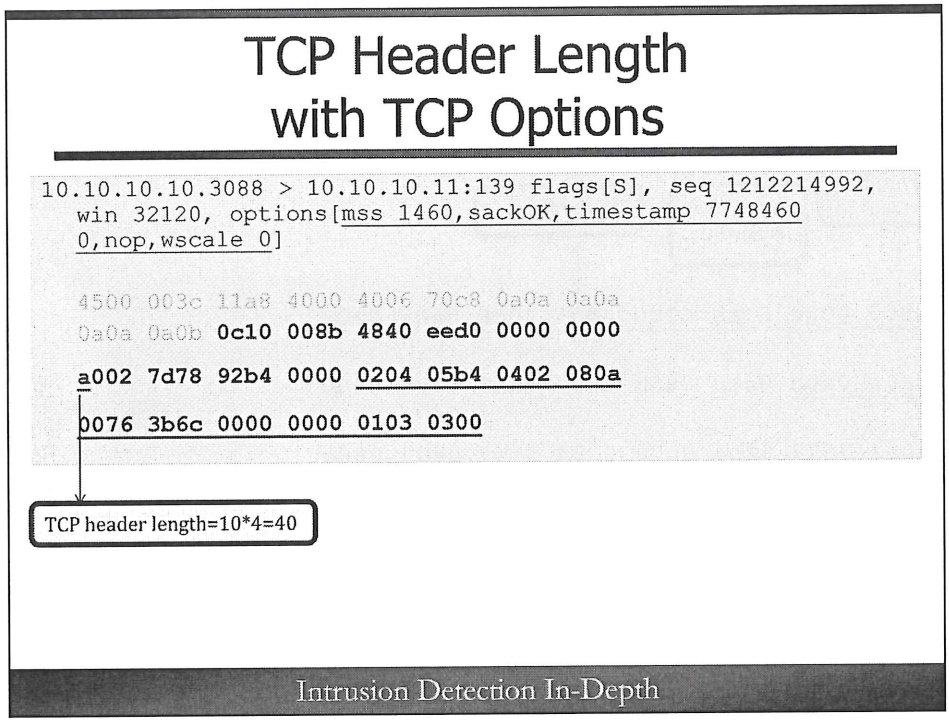


There is a single length field in the TCP header. This represents the length of the TCP header itself. Like the IPv4 header, the TCP header can have options. The TCP header is 20 bytes long without TCP options. Unlike IPv4, TCP options are very common and more often than not used in TCP sessions. In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



Examining a real datagram, we 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, we 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.

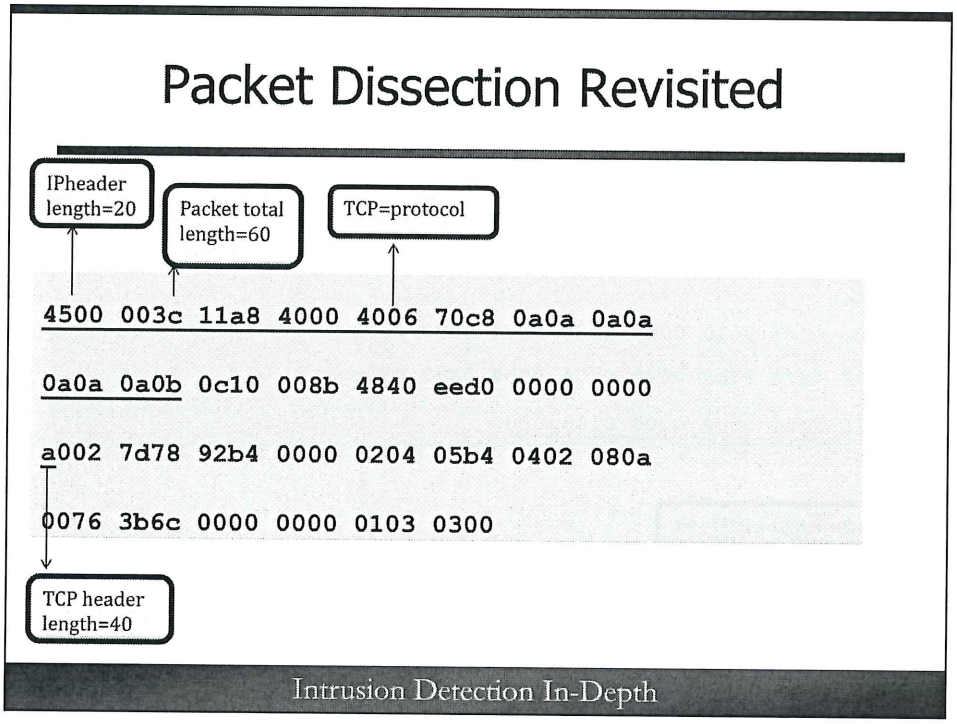


Now look at the above datagram. You see that it has a TCP header length of 0xa which is a decimal 10. This value multiplied by 4 indicates a TCP header length of 40 bytes. The underlined portion of the above output represents the TCP options values. These include options maximum segment size of 1460, sackOK, timestamp and a wscale (window scale). 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 since it stands to reason that might be the best place to pad. Different operating systems place the pad bytes before or after specific options. In fact, 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.

These options need to be stored in the TCP header after the 20 bytes of standard TCP header values. We'll examine the meaning and purpose of these options later.

This is a SYN packt that is why there is not data

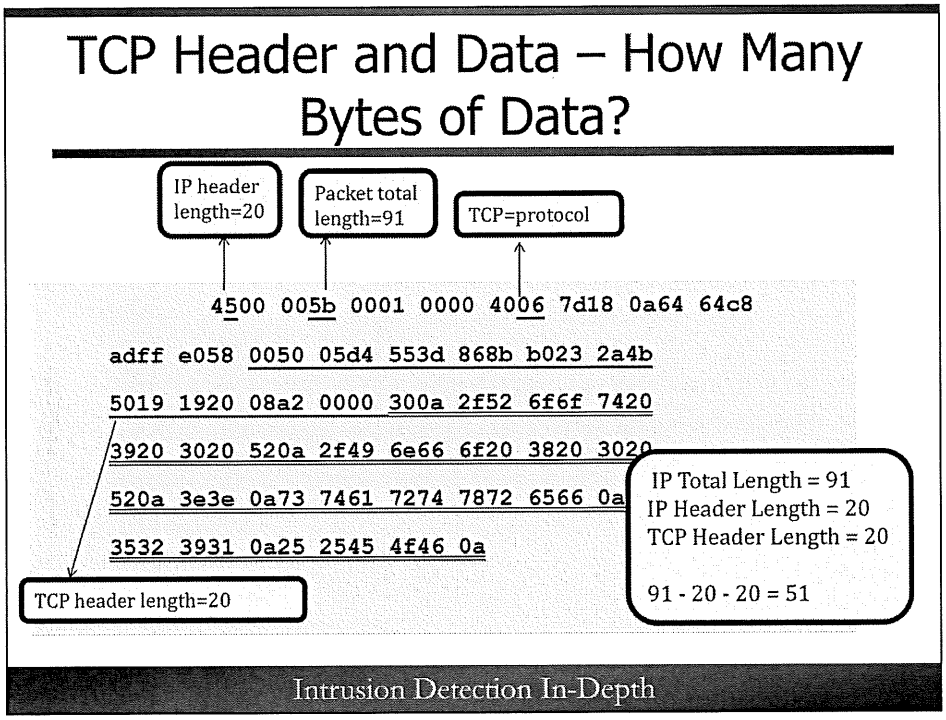
20 IPv4
40 TCP



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 we learned about the IPv4 header and add our new knowledge about the TCP header to interpret this hexadecimal.

First, we know 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, we know that 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, etc. 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 which is why there is no data.



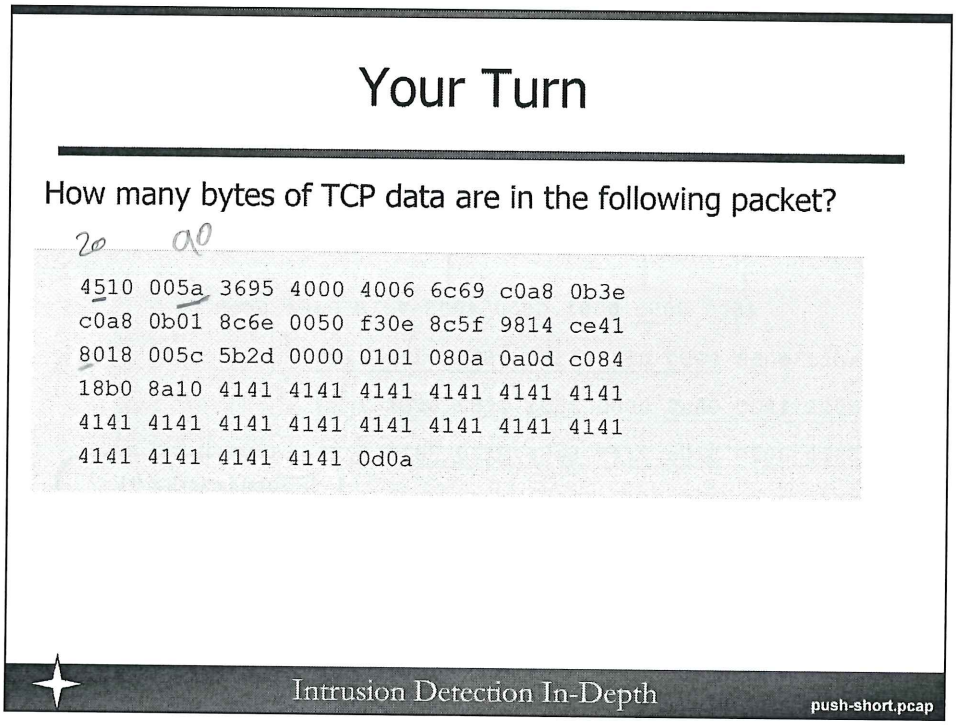
Now that you've had a chance to dissect some packets on your own, you understand the need for the length values. They indicate where given protocols, header, and values begin and end. This is essential for you to dissect the packets just as it is for the TCP/IP stack to de-encapsulate those same packets.

Remember on the first day that 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 datagram, and the TCP header. Yet, there is no dedicated field for the TCP data length. You might argue that this isn't needed since all the bytes after the TCP header are data bytes. But, for us to validate this, we can use a derived length. The formula for the number of TCP data bytes is:

IP total datagram length – IP header length – TCP header length = TCP data bytes

Therefore, using the packet above we have:

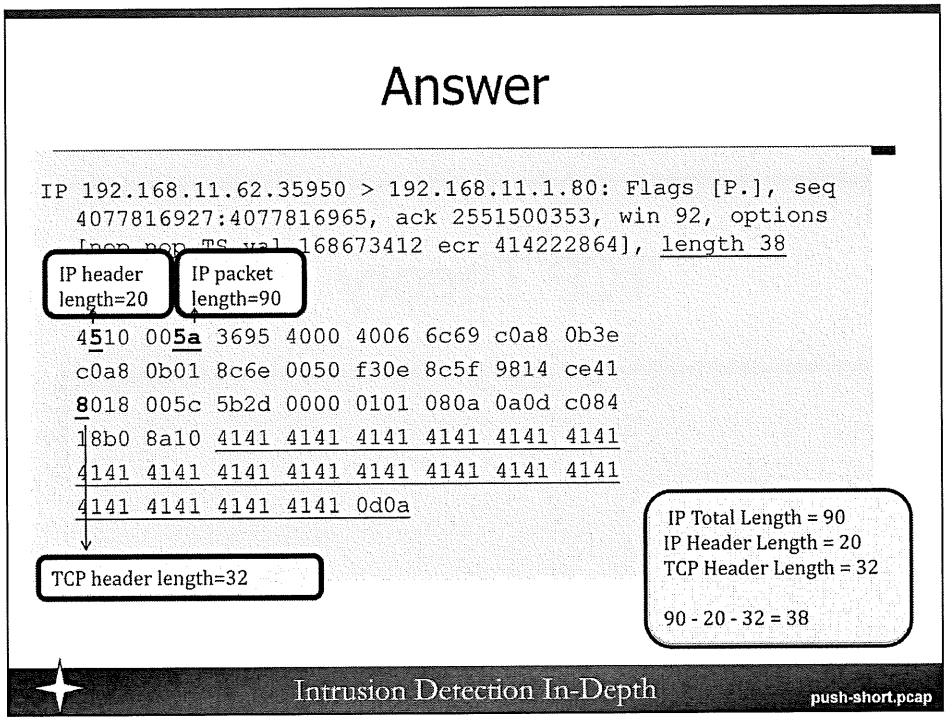
91 – 20 – 20 = 51



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

Once you are done, you can verify your answer by entering the following command:
tcpdump -r push-short.pcap -nt

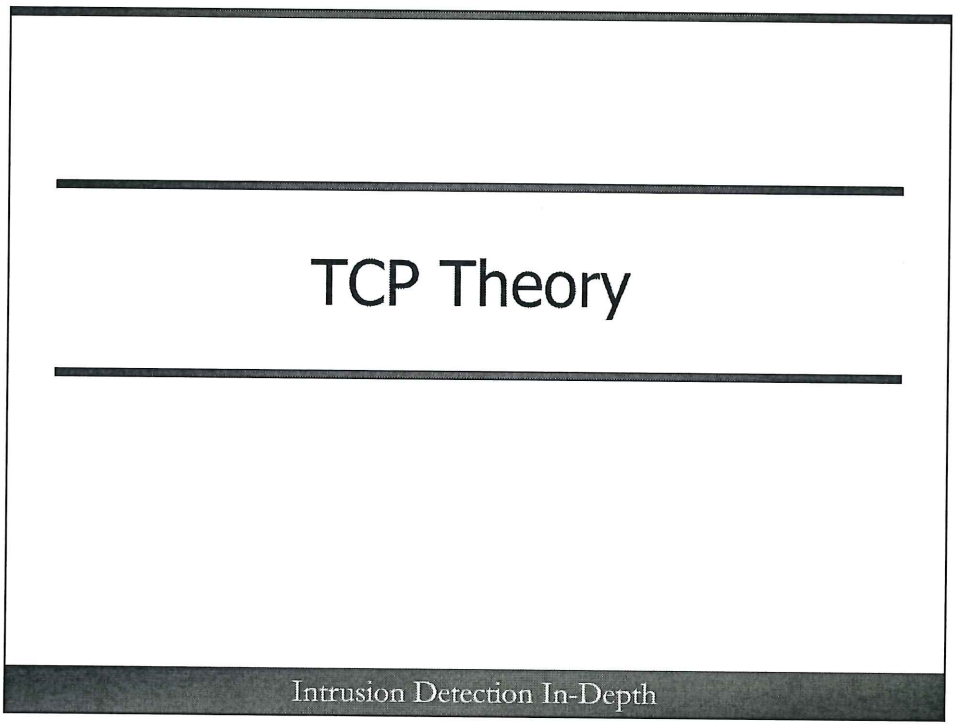
The length value appears in tcpdump, reflecting the number of bytes in the TCP data.



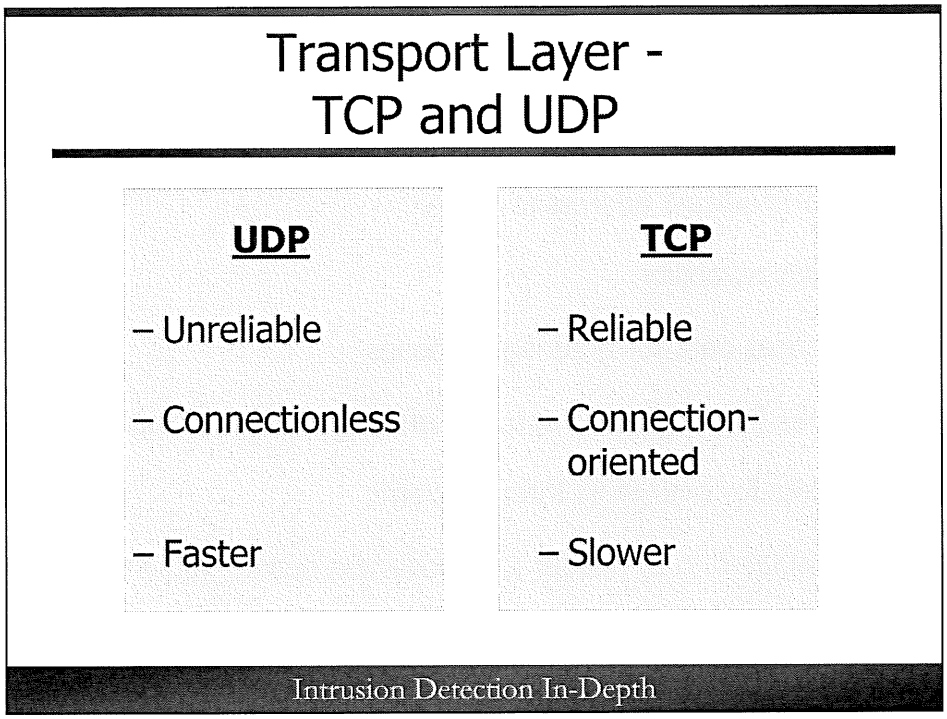
Tcpdump indicates that there are 38 bytes of data. There is a standard 20 byte IPv4 header, a total packet length of 90 and a TCP header with options 32 bytes in length. Therefore, our formula indicates the presence of 38 TCP data bytes. If you count the underlined data bytes, there are 38 bytes present.

You can verify your answer by entering the following command:

tcpdump -r push-short.pcap -nt



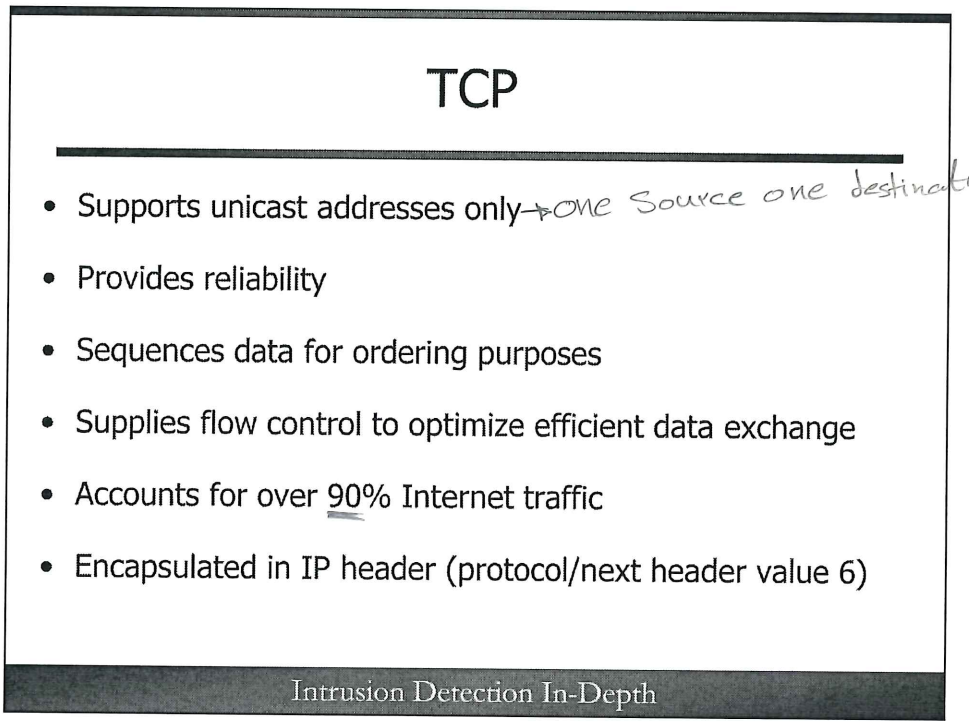
This page intentionally left blank.



Let's examine the difference between TCP and UDP before we begin to look at TCP. To review, the transport layer is responsible for end-host to end-host communications.

TCP is a connection-oriented protocol. Connection-oriented is just what it sounds like -- the software does everything that it can to ensure the communication is reliable and complete. There is a lot of overhead involved in ensuring that there is reliable delivery. TCP will ensure that any lost packets are retransmitted.

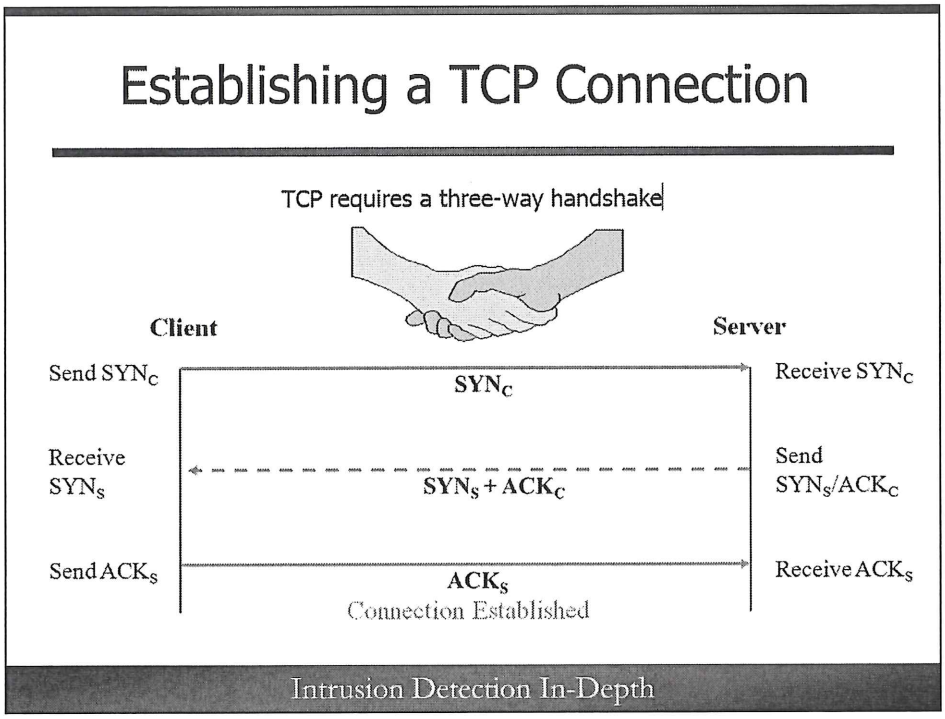
On the other hand, UDP makes no guarantee of reliable delivery. If a packet is lost, UDP won't care. There is no notion of connection or state and UDP is faster because it doesn't have to guarantee reliability.



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 destination host acknowledging all received data. This is the only way that the sender knows that the data has been received.

TCP also gives each segment a sequence number. The sequence number represents the first byte of data in each TCP segment sent. Since TCP segments are transported in IP packets and since 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 re-ordering 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.



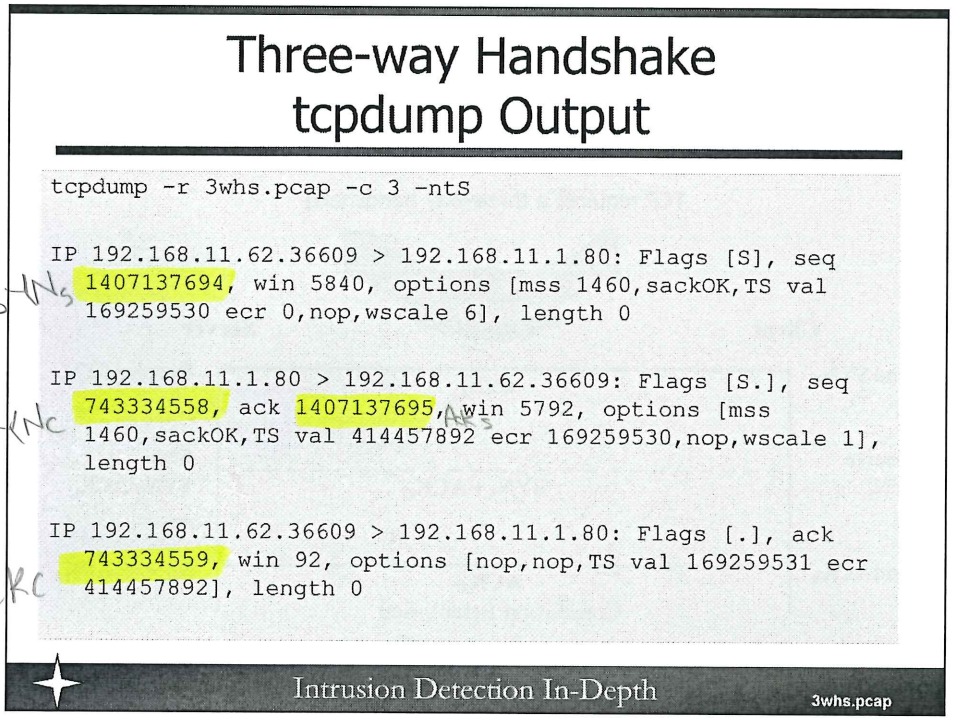
You see that establishing a TCP connection is almost ceremonial in nature, involving what is commonly known as the three-way handshake. This is normally done before any data can be passed between the two hosts. What is depicted is the client or source host initiating a connection to the server or destination host. Because this is TCP, a listening port or service must be identified on the server in order to connect. As well, the client must designate a port for its communications.

For the TCP three-way handshake, the client first sends a SYN flag (SYN with a subscript of c) set 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 a connection request of its own with the SYN flag set (SYN with a subscript of s) to the client and acknowledges the client connection request with an ACK (ACK with a subscript of c), all in a single packet. 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 (ACK with subscript of s) to the server acknowledging it has received the server's SYN attempt. Once the server receives and accepts the ACK, the connection has been established. Data can now be exchanged between the two. Once the connection is established, every segment should have the ACK bit set. As much as possible, ACKs are "piggy-backed" 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 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, the "SYN" that we refer to as the flag to start a TCP connection is actually an abbreviation for synchronize sequence numbers.



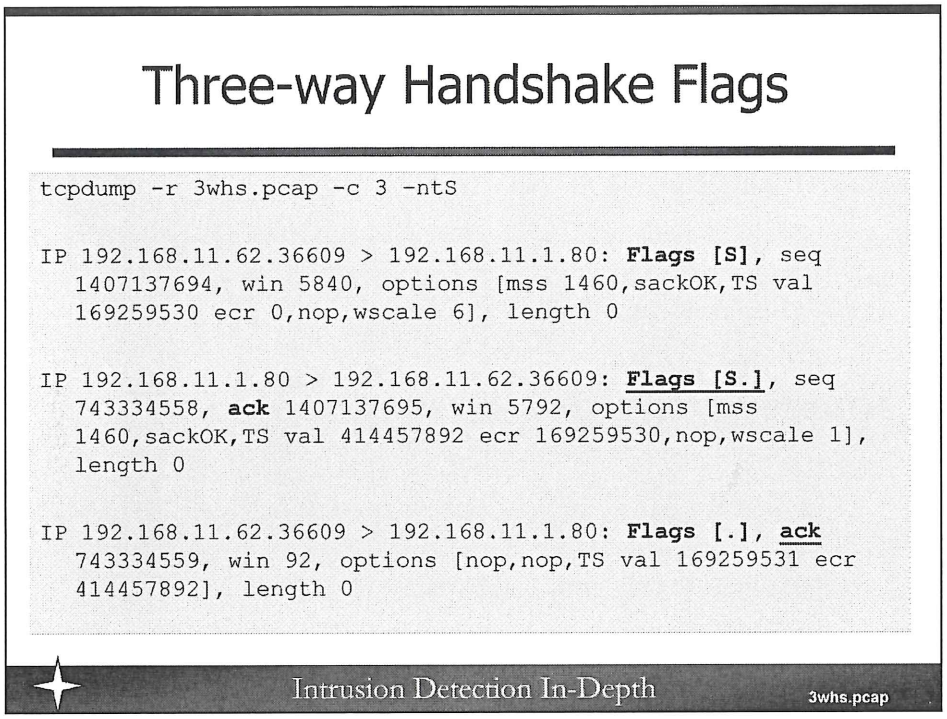
This is the way that tcpdump shows the packets involved in the session establishment – also known as the three-way handshake. Here is an instance where we need a visual overview of the packets and tcpdump serves the purpose better than Wireshark since Wireshark would require us to examine each packet individually, perhaps confusing the nature and sequence of what is transpiring.

We'll use these same sequences of packets to describe different facets of TCP in the next several slides.

To follow along, enter the following command:

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

This command reads from the file 3whs.pcap – only the first three records – and suppresses name resolution and tcpdump timestamp output. A new introduced option is the –S switch. This shows TCP sequence and acknowledgement numbers as absolute rather than relative values. We'll discuss that in more detail. The reason this switch is required is that tcpdump, by default, shows relative sequence and acknowledgement values, making it difficult to follow if you are not used to the syntax.



First, let's look at the TCP flags involved in the three-way handshake. We'll discuss the various flag values in more detail in upcoming slides. TCP flags are used to denote the state of the TCP connection and the appropriate exchanges of TCP segments in that given state. The SYN flag is used to synchronize or establish a session. The ACK or acknowledgment flag indicates the acceptance of data. This sounds unusual in conjunction with the three-way handshake since 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 the RFC 793 pertaining to TCP.

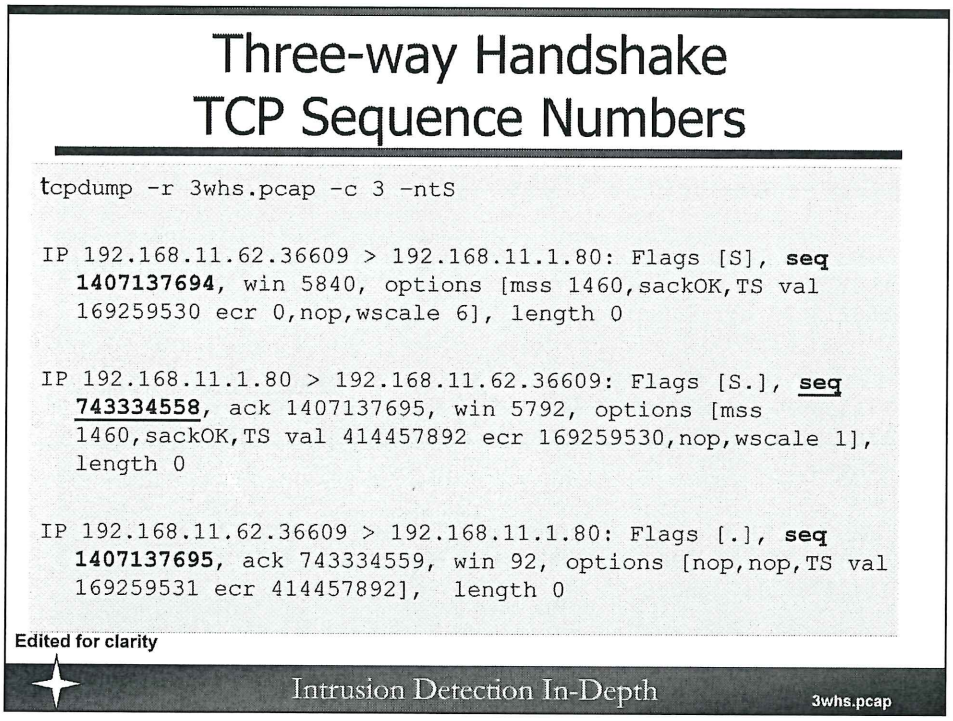
Above, host client 192.168.11.62 wishes to open a session with server 192.168.11.1 on port 80. The SYN flag is set on the initial packet of the three-way handshake. If the server listens on port 80 and is able to open a new connection, it responds using the ACK flag that it received the client's request. As well, it must also indicate its intention to open a new connection or socket using a SYN flag with client 192.168.11.62. Remember TCP is full-duplex and both sides of the connection need to initiate a session with the other side.

The last packet of the three-way handshake is where the client host must acknowledge using the ACK flag that it received the server's request to initiate a connection with the client.

Tcpdump has some unique syntax. For instance, there is a "Flags[S]" notation to specify that the SYN flag is set. Yet, even though the ACK is a flag, it elects to place it elsewhere with an acknowledgement value. And, in the last packet, the flags are set as "Flags[.]". This means that none of the following flags is set: SYN, FIN, RST, PUSH, ECE, or CWR. Don't worry about the purpose of those flags just yet.

To follow along, enter the following command:

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



Let's look at these same three records, but in terms of TCP sequence numbers. TCP sequence numbers assign a chronological 32-bit number to a given side of the conversation. These are used so that the receiver knows how to sequence these packets when they arrive since they may not arrive in the same order as they were sent.

The first TCP sequence number selected by each side in the exchange is known as the Initial Sequence Number or ISN. It should be a random number. Each side of the conversation generates its own ISN. Each side sends a different set of TCP segments and each side has a unique chronology of segments sent. Every segment in a TCP exchange must have a TCP sequence number value.

In the above exchange, the client has an ISN of 1407137694. The server has a different ISN value of 743334558. All subsequent segments sent by the client have sequence numbers relative to its ISN. Similarly, all subsequent segments sent by the server have sequence numbers relative to its unique ISN.

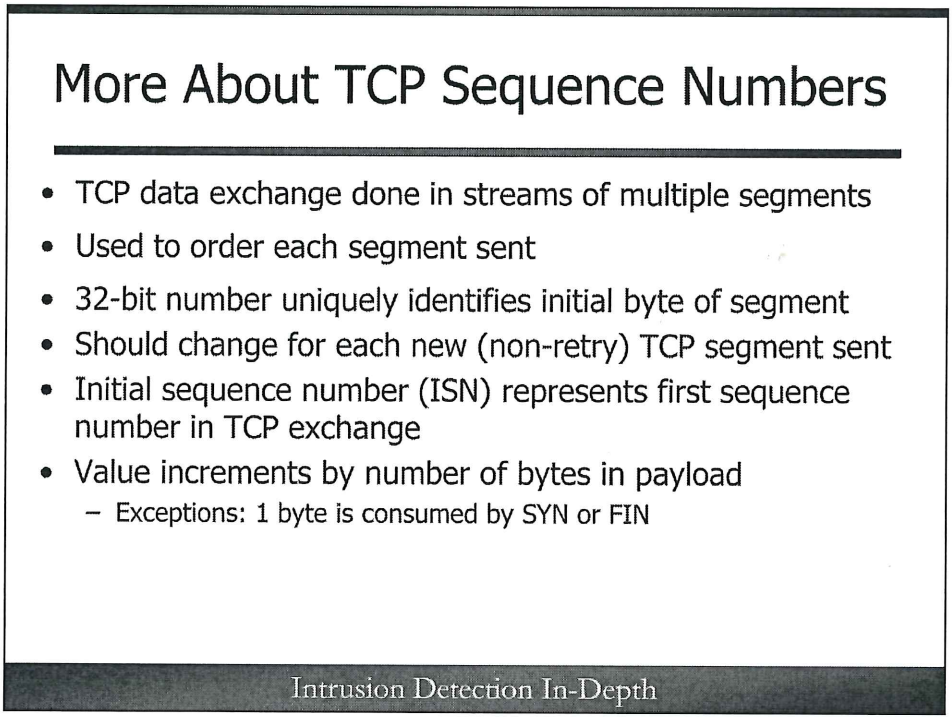
The sequence numbers increment by the number of payload bytes sent in a given segment. If no payload bytes are sent by a given side when sending a packet – say an acknowledgement packet – the sequence number on that segment should be the same as the previous segment's sequence number. There are two exceptions to this when segments contain either a SYN or FIN flag. Though rarely seen on a segment with the SYN flag set, and occasionally seen on a segment with the FIN flag sent, neither should have payload, although data payload is allowed.

To follow along, enter the following command:

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

Per RFC 793, the SYN flag consumes 1 sequence number, although there are no data bytes. This is standard convention, though tcpdump doesn't make it easy to understand this with its display of a length of 0. As we've seen in our exploration of the derived number of TCP data bytes, tcpdump typically annotates the length of the data bytes after the record.

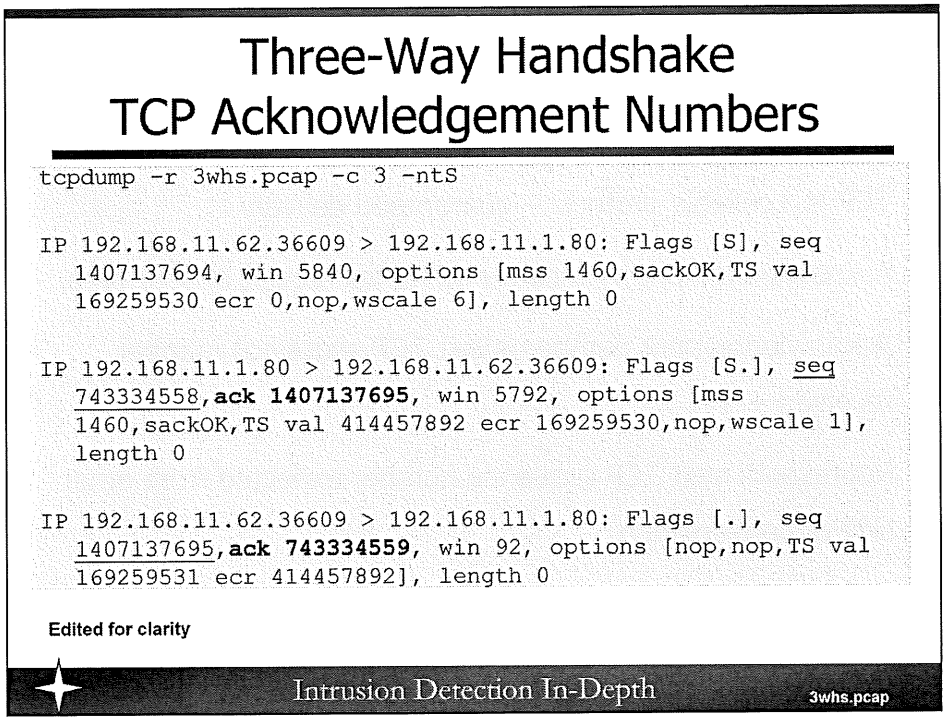
The FIN flag consumes a sequence number too per RFC 793 even if there is no data accompanying the segment.



When data is exchanged on TCP, it is very likely that it will have to be done using many different packets each with its own TCP segment. As we know, TCP is a reliable protocol. As such, it must be able to know when packets have not been delivered. 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. As we've seen when a TCP session is started, each side of the session will select an ISN as the first sequence number.

These TCP sequence numbers give the entire stream associated with the TCP exchange a notion of order. Since each TCP header contains a sequence number, a packet that is lost is easier to identify. 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 ordinarily consumed with each byte of payload data. For instance, if the payload contains 1,000 bytes of data, the TCP sequence number will increment by 1,000. There are two exceptions to this rule. The first is that per RFC 793, one TCP sequence number is consumed by the SYN. In other words, the ISN value increments by 1 on the TCP sequence number of the ACK of the three-way handshake. You also see the ISN increment of 1 byte when the server returns the SYN/ACK. The acknowledgement value is a value of 1 more than the client's TCP sequence number. The second exception is that this same concept applies to any packet where the FIN flag is set. A FIN also consumes 1 TCP sequence number. Most often, the FIN has no data payload. More often, the SYN has no payload. However, if there is accompanying payload, the TCP sequence number increments by the number of bytes in the payload + 1 byte consumed by either the SYN or FIN.

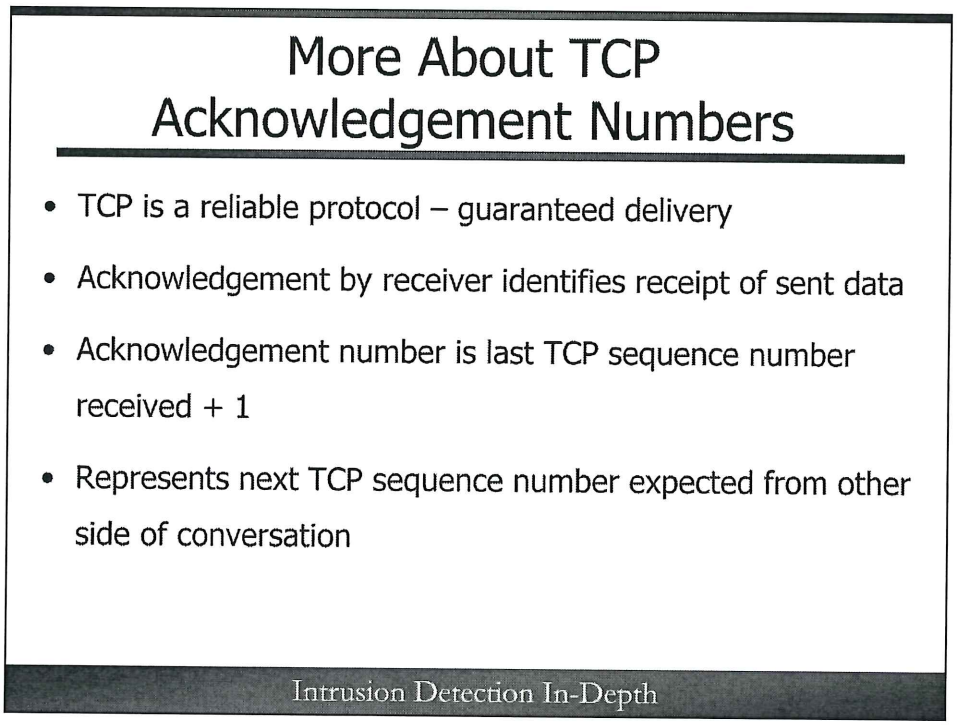


Now, let's reexamine these same three records, but with a focus on TCP acknowledgement numbers. The acknowledgment number is processed only when the ACK flag is set, otherwise the value is ignored. We'll talk more about TCP flags in an upcoming slide. As TCP sequence numbers assign a chronological 32-bit number to a sender's TCP segments, the receiver returns an acknowledgement number indicating whether or not 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 or not a segment arrived or not.

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

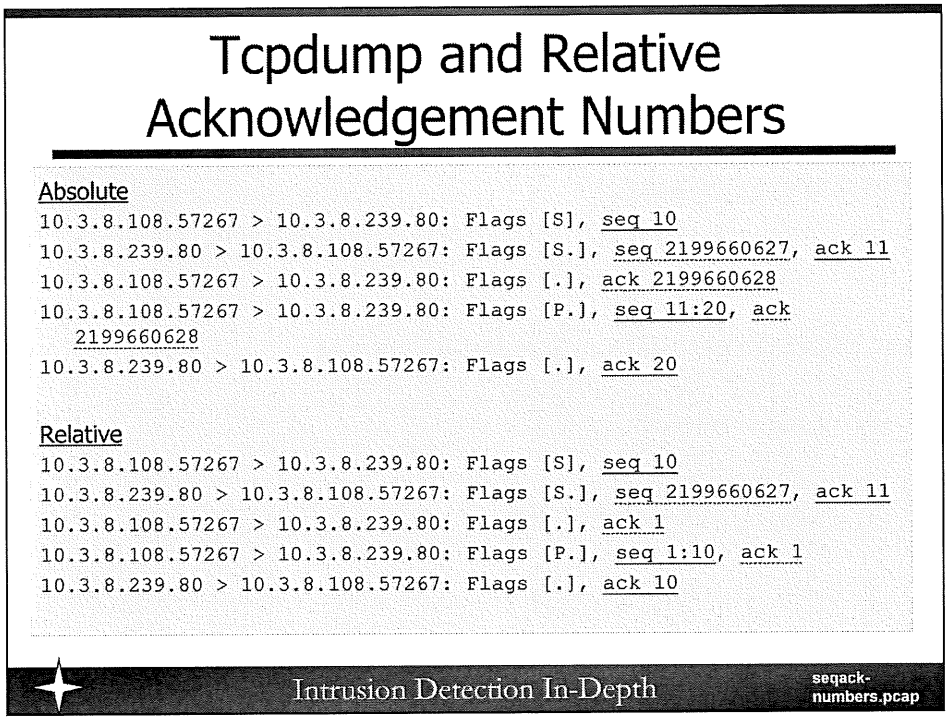
To follow along, enter the following command:

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



Because TCP guarantees reliable delivery of all data, it needs some kind of mechanism to know if TCP data has been received. This function is left to the receiving host; it must acknowledge receipt of data. It does so by first setting the acknowledgement flag in the TCP header. It is not enough to simply do an acknowledgement that data has been received. It must acknowledge what data has been received.

If a packet is lost from receiver to sender, the receiving host will get 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 returned to the sender for the sender to become aware that it needs to retransmit the lost packet.



Here is the same sequence of packets displayed using tcpdump – the first set uses absolute sequence numbers, the second set relative. The output has been truncated to show the pertinent details. Also, the client side of this session is crafted. The reason this is mentioned is because an ISN of 10 is abnormally small.

By default, tcpdump shows relative sequence numbers because 32-bit sequence number values may clutter the output. Relative numbers take up far less visual space. All sequence and acknowledgement numbers above that are related to the client have a solid underline. Those relative to the server have a dashed underline.

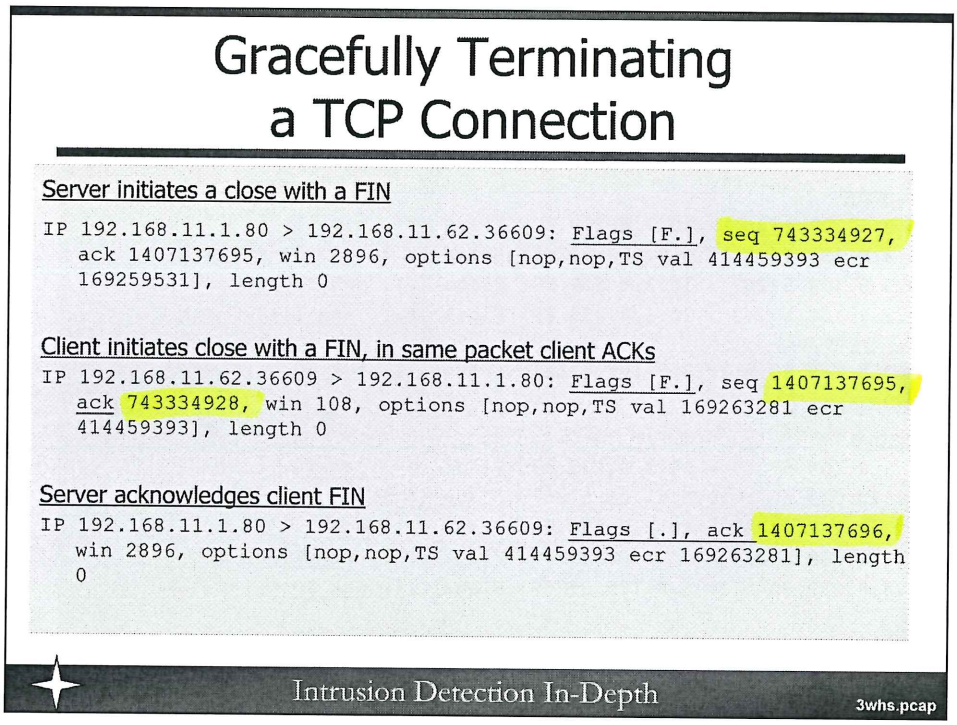
The ISN of the client is a crafted abnormal value of 10 and the true ISN value of the server is 2199660627. The first session with absolute values shows the sequence numbers and acknowledgement numbers as they are found in the TCP header. Look at the second session with relative values where tcpdump 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 2199660628 = 1 + 2199660627 (server's ISN). Similarly the beginning sequence number of the fourth packet set by the client is 1 and spans 10 bytes. This is relative to the client's ISN of 10. Accordingly, in absolute value this is 11:20 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 at all, 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/acknowledgment numbers:

tcpdump -r seqack-numbers.pcap -ntS

tcpdump -r seqack-numbers.pcap -nt



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:

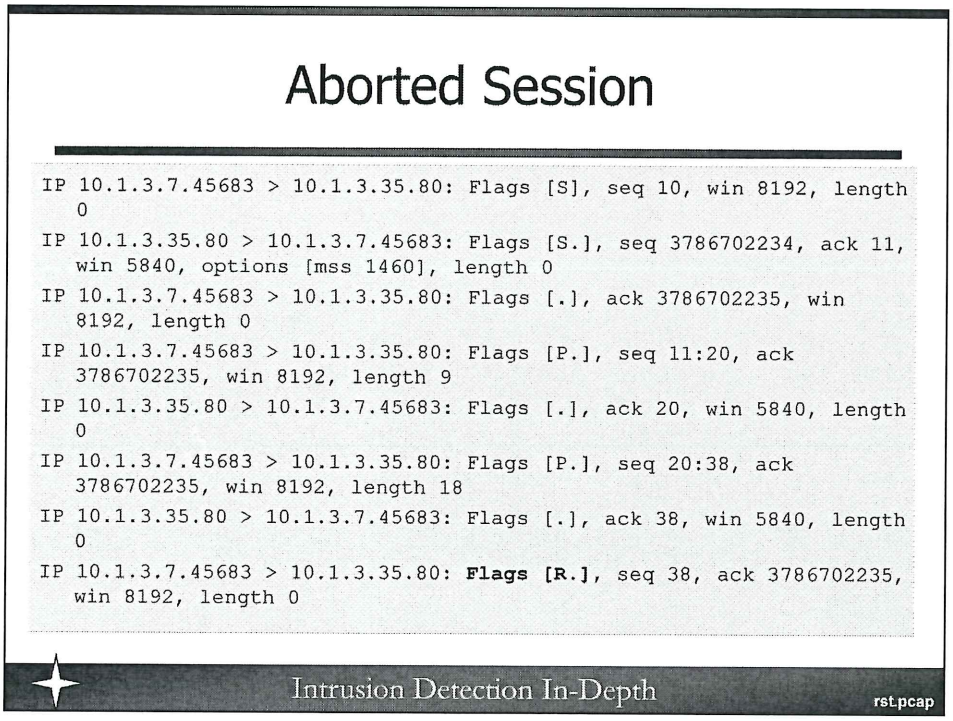
Side 1 initiates a FIN to close its side
Side 2 ACKs the FIN
Side 2 closes its side with a FIN
Slide 1 ACKS the FIN

In practice this may or may not happen. In fact, in the above output steps two and three 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 the sequence above. 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, however 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 clients close, issuing sequence 1407137696 back to the client's FIN.

If you'd like to follow along, enter the following command:
tcpdump -r 3whs.pcap -nt (last three records)

Neither the client nor 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 presents. A segment with a FIN flag also consumes a sequence number.

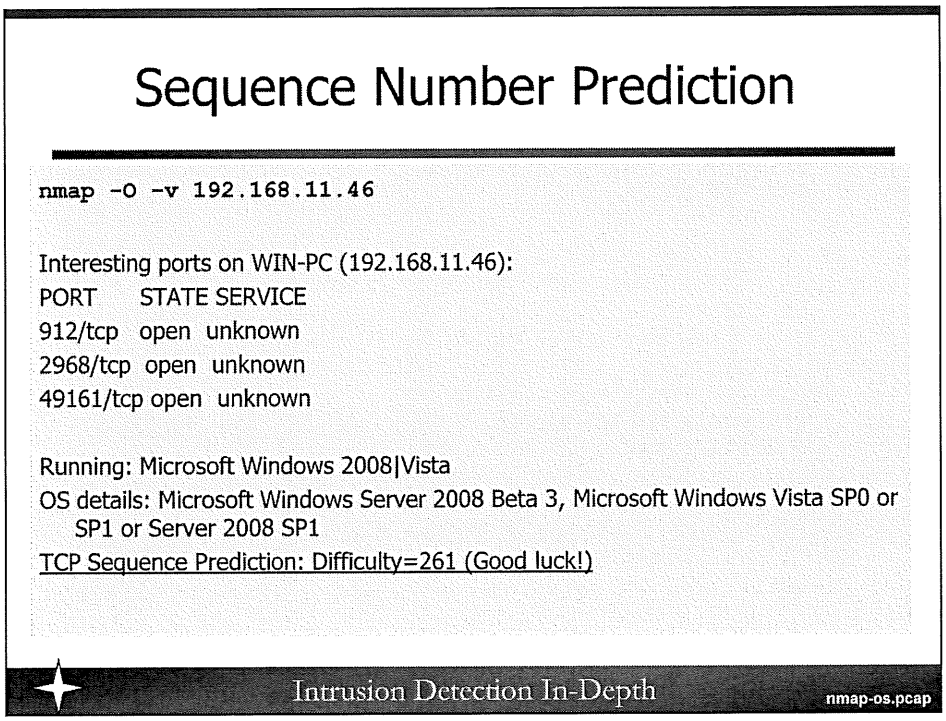


An abrupt close of an existing session can be accomplished using a RST flag. One thing you need to understand about a RST flag is that the associated packet is the last in the session. There should be no subsequent segments sent. This needs a caveat – this assumes that the segment carrying the RST flag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

You will also see a RST flag in a response to the sender when a receiver gets a packet to a non-listening port or when the segment is not part of an established session. For instance, let's 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 will issue a RST since it knows nothing about the session and has no that indication that it sent an outbound SYN to the remote host.

To view the output, enter the command:

tcpdump -r rst.pcap -nt



The formula that TCP/IP stacks use to select their initial sequence number is examined by Nmap to help fingerprint the operating system. There is a file that comes with Nmap, nmap-os-db, that has a list of different operating systems and expected returned values for a set of tests that will be executed by Nmap against a target host. Nmap is able to categorize a particular operating system by matching received values of different normal and abnormal stimuli after the scanning host with expected values for a given operating system.

One test executed by an operating system fingerprinting Nmap scan examines the initial sequence numbers generated on the target host from sending connections to a listening port. Different TCP/IP stacks use different formulas to generate the ISN. Some very old operating systems used a predictable increment for the ISN for each new connection. But someone watching and sniffing could hijack a connection using this information as was done in the infamous Mitnick attack. Other older operating systems had a time-dependent formula that predictably increased the ISN based on a given time change. This too is not considered very secure.

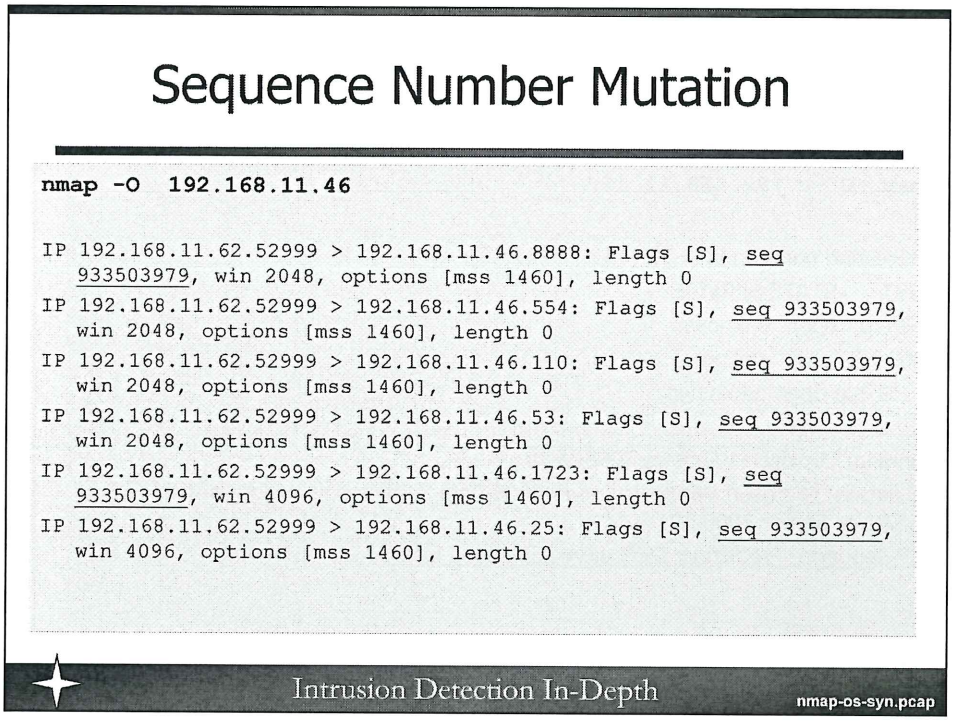
Nmap sends a series SYN packets to an open port to observe the remote OS pattern of ISN value generation in the returned SYN/ACK. It uses a complex algorithm to determine the difficulty of predicting those ISN numbers and assigns a predictability value. The higher the value upwards to 9999999, the better. However, any assessment of "Good luck!" as above is considered difficult regardless of the value. According to the Nmap source code – the categories and values are: "Trival Joke" where the difficulty value is less than 3, "Easy" – less than 6, "Medium" < 11, "Formidable" < 12, "Worthy Challenge" < 16 and everything greater is assigned a difficulty of "Good luck!".

If you care to understand more about the formula used to assess ISN predictability see:

http://nmap.org/book/osdetect-methods.html#osdetect-gcd

To view the output, enter the command:

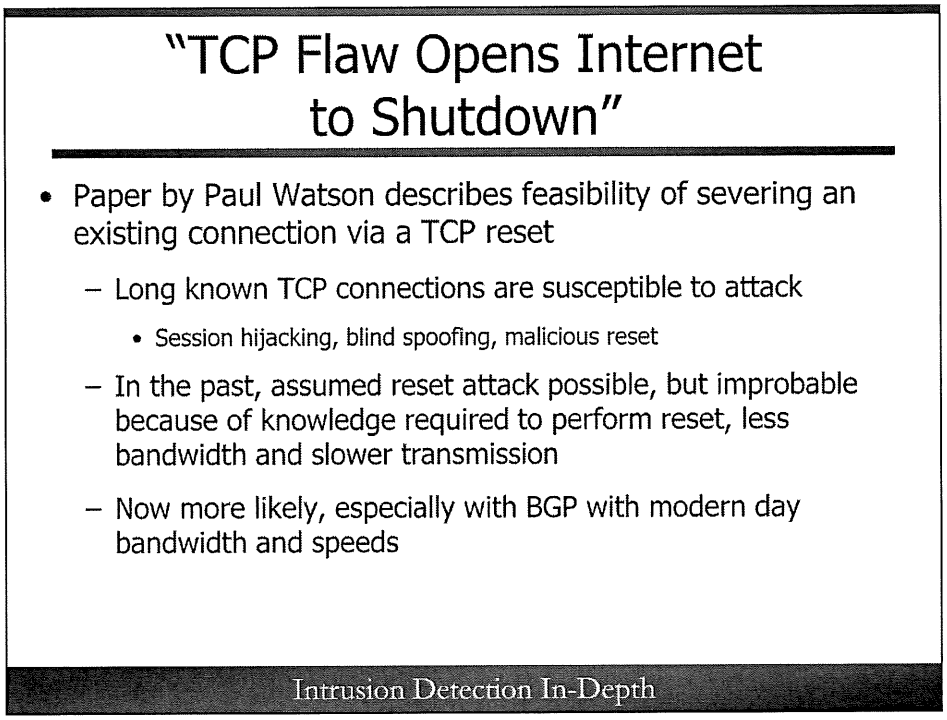
tcpdump -r nmap.os.pcap -nt



When Nmap attempts to assess a remote host's operating system, it has some unique identifiable characteristics. One of those is the repetition of the ISN from the scanning host. A client host should use a unique ISN when attempting to connect to different ports of the remote host. This is a small excerpt from the scan where dozens of other packets have this same ISN. This is a good way to identify an OS detection or other scans, such as a SYN scan, from Nmap.

To view this output, enter the command:

tcpdump -r nmap-os-syn.pcap -nt



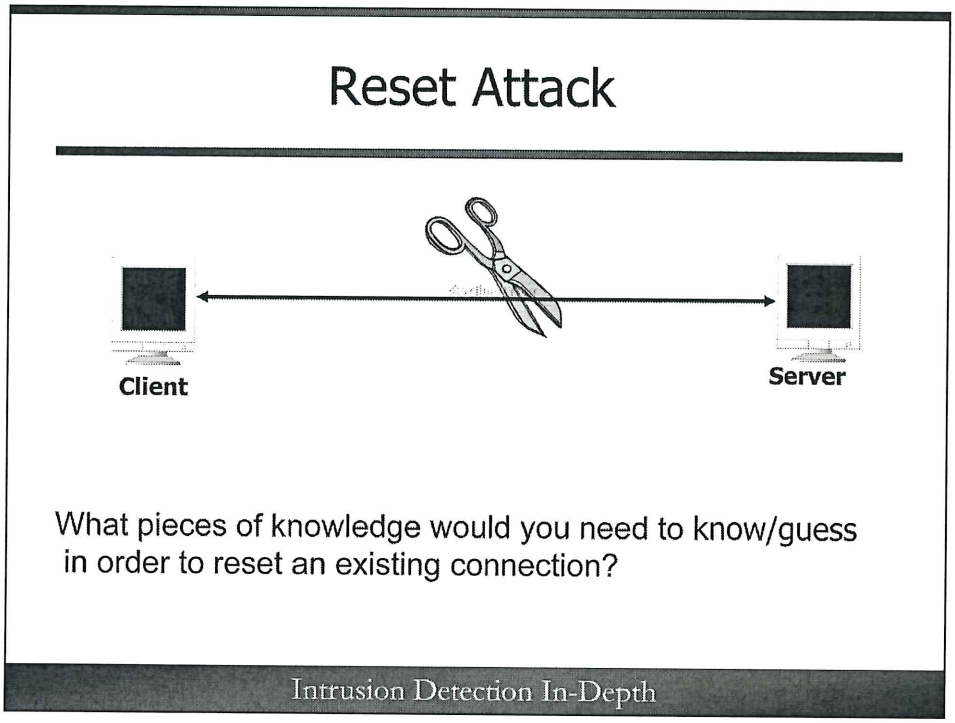
There is an interesting paper that Paul Watson released "Slipping in the Window: TCP Reset Attacks". We can examine this to reinforce the knowledge we've gained about TCP.

Watson delivered a presentation at CanSec West about his paper. The general media got wind of the paper and, in their normal calm and deliberate manner, sensationalized the heck out of the story without really understanding the underlying principles. It is understandable that they didn't grasp the concepts of the subject – the feasibility of a TCP reset attack taking down the Internet since it is pretty technical. Yet, that didn't stand in their way of reporting the doom of the Internet.

It has long been known that TCP may not be able to withstand attacks. In the past, when TCP initial sequence numbers were not randomized, TCP was subject to attack by blind spoofing employing a reasonable guess of a likely generated initial sequence number. This is all because TCP was envisioned and created in 1981 when security wasn't an issue and the intended use for TCP and other protocols was not a global-facing hostile environment. Times have changed and we've seen TCP come under scrutiny and attack since its implementation.

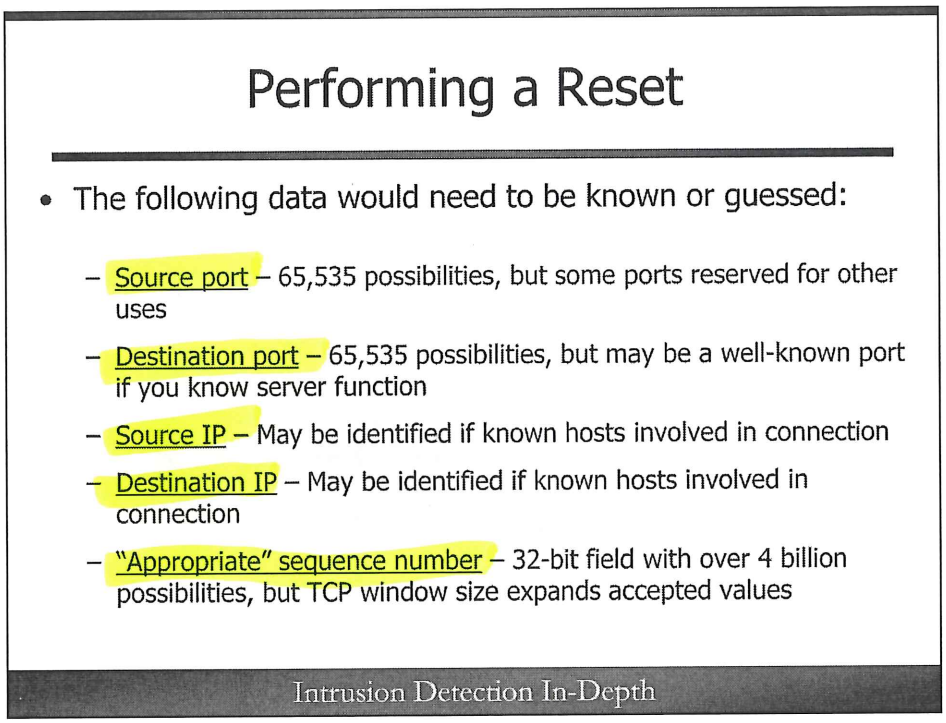
It has also been known that TCP is susceptible to an attacker severing an existing connection using a reset against either the client or server. This was something far more likely to happen if the attacker could somehow sniff the traffic and learn the information necessary to reset the connection. It was considered infeasible that a connection could be blindly reset since this would require too much guesswork. In his paper, Watson describes the current types of media and bandwidth available to accomplish successful reset attacks that may be able to abort existing connections in a matter of seconds or minutes.

The paper can be found at:
http://www.osvdb.org/ref/04/04030-SlippingInTheWindow_v1.0.doc



Knowing what you do about the way TCP works, what information about the connection would you need to gather or guess to craft a packet to reset an existing connection? An appropriate reset can be directed to either the server or client side of the connection and still terminate the connection.

A reset is easier to use to sever a connection than a FIN since the reset requires a single packet to be spoofed with no expected communication after it. A FIN would need much more interaction, requiring the attacker to generate the FIN, wait and listen for the server's FIN, and finally acknowledge the server's FIN.

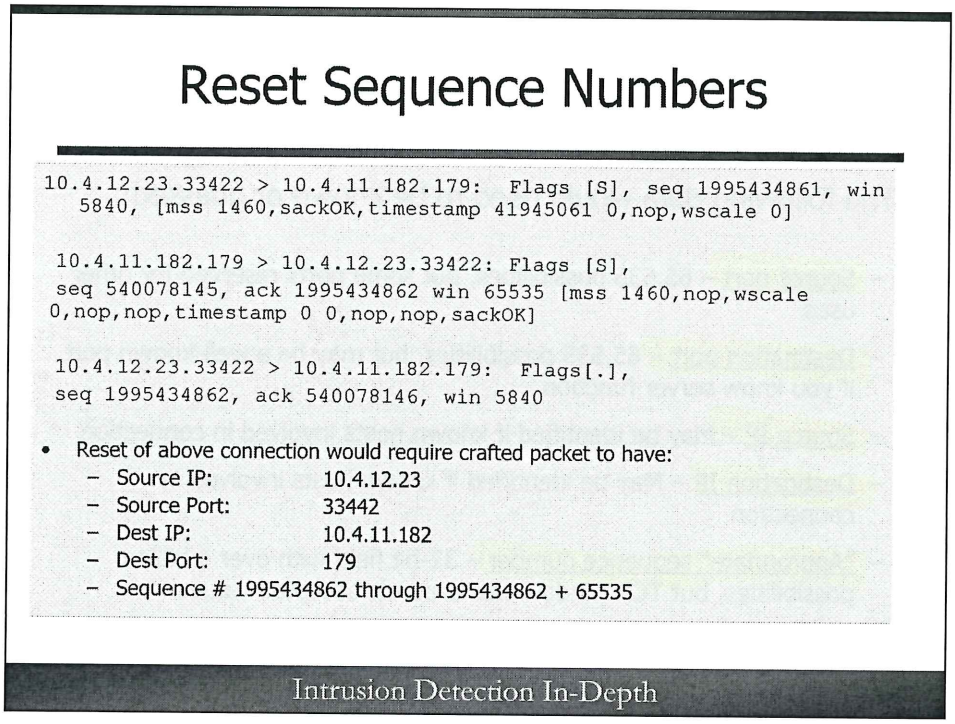


Let's talk about an existing connection that is the target of a reset attack in terms of client to server. Let's focus on an attack simulating a client reset to the server. The sender of a crafted reset packet will have to know/guess the source port in use of the target session. This is a 16-bit field, but many operating systems reserve a number of ports and not all 65535 will be used as ephemeral ports. The destination port is typically a well-known port – for instance port 179 for Border Gateway Protocol (BGP). And the source and destination IP addresses must be known. Finally, a crafted packet has to have an appropriate TCP sequence number for that connection. We'll discuss what is appropriate in just a bit.

This sounds like a an impossible task if this data cannot be sniffed. But, there is one protocol – BGP that may be at most risk. That is because BGP is often used between backbone routers that can be discovered by an attacker using traceroute to identify the IP addresses of the connection. Also, we know the destination port is 179. That just leaves the source port and the sequence number as unknowns. And, that is another reason that BGP is a good target, it maintains a persistent connection of activity. In other words, sessions are established, data is exchanged, and the connection is maintained for quiet periods. Guessing an appropriate sequence number when data is being exchanged back and forth becomes more difficult because you are attacking a moving target. If the connection is relatively quiet or has periods of inactivity, it is easier to attack. BGP uses persistent connections so that new connections don't have to be established each time data is exchanged.

It turns out that an "exact" TCP sequence number is not necessary to reset a connection. In fact, RFC 793 explains that the TCP sequence number need only be in the range of the sender's previous sequence number plus the number of bytes in the receiver's announced window size. As you know, a host announces its initial window size on the initial SYN connection. Different operating systems select different TCP window sizes. An operating system with a larger window size has a better chance of having the connection reset.

How is this different from the TCP sequence required when data is exchanged in an established session? The sender must specify an exact TCP sequence number that indicates the next expected sequence number by the receiver – that is a value of 1 more than the last TCP sequence number received.

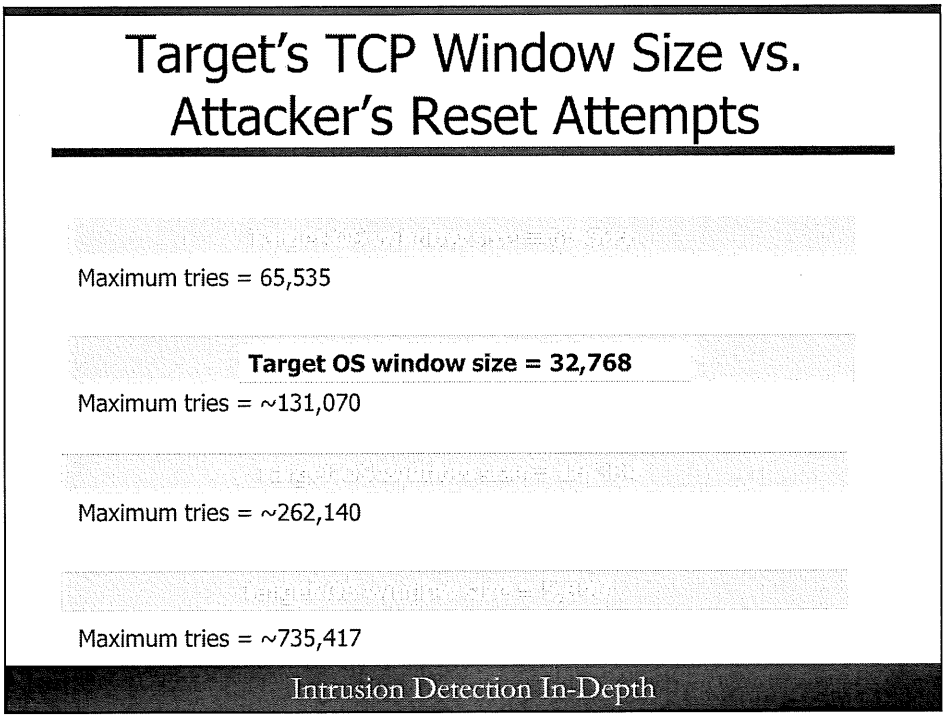


Take a look at the tcpdump output of an existing BGP connection and let's examine the crafted packet required values to reset the client-to-server side of the connection. For discussion sake, we are using the reserved network 10.x.x.x, but a real attack on a publicly available network would not use a reserved address.

The attacker would have to identify the source IP as 10.4.12.23 and would have to identify or guess the source port as 33442. Additionally, the packet would have to reflect the destination IP of 10.4.12.182 and the destination port for BGP of 179. Finally, the sequence number of 1995434862 (next expected client sequence number) through 1995434862 + 65535 (server's TCP window size) would have to be guessed. If a crafted packet contains all these fields, and there has been no or little data exchange between the actual client and server, the connection will be reset.

Basically, if an attacker can create a program or use an existing packet crafting utility to generate many packets with different combinations of source ports and sequence numbers, it may be possible to reset the connection. This assumes that the attacker has been able to identify the source and destination IP addresses of hosts exchanging BGP information.

The case where the target host window size is the largest possible (without using the TCP option window scale) of 65535 bytes provides the best opportunity for attack. Since the TCP sequence number is 32 bits, this means that there are 65535*65535 possible ranges of sequence numbers that will fall in the appropriate TCP window range. The worst case scenario would be that the appropriate sequence number would be discovered on the last (65535^{th}) try. Remember too, if the source port is not known, it will have to be guessed along with the sequence number and this leads to many more combinations of guesses. In fact, if we assume all ports can be used as ephemeral source ports, this means we're back to over 4 billion combinations. However, as Watson notes in his paper, many times the source port numbers hover around the 1024 and greater range.

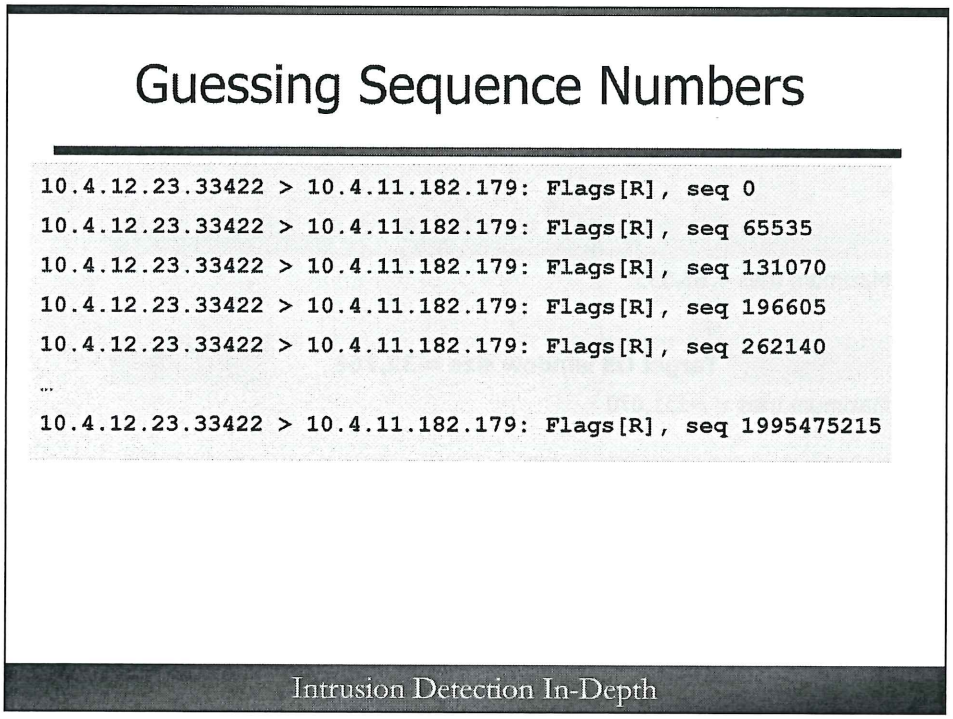


For the moment, let's assume that the attacker has identified all other required values about an established connection except the TCP sequence number that must be used to reset the connection. As you can see the larger the target's TCP window size, the fewer number of maximum tries or guesses the attacker has to perform to successfully reset the connection.

The maximum TCP window size is 65,535 and is typically found on Windows 2000 and later Microsoft operating system hosts. It would require 65,535 attempts to guess the correct sequence number where we've already mentioned the maximum number of sequence numbers is over 4 billion. Next, a target operating system that has a window size of 32,768 such as found on HP-UX 11 would require twice the number of maximum attempts. Cisco operating systems may have a window size of 16,384 and would require over 262,140 maximum attempts. Finally, a Linux 2.4 kernel host has a default TCP window size of 5,840, necessitating a maximum 735,417 attempts to reset the connection.

Realistically, each of these maximum number of attempts would have to be multiplied by the number of source ports that a given client may use. As discussed, this may be less than the 65,535 available since some are reserved. Depending on the target and simulated client, this may take hundreds of thousands or even several million connections to get right assuming that the attacked connection is relatively dormant.

A target host that uses the window scale TCP option augments the windows size - hence the range of acceptable TCP sequence number values. This may greatly reduce the number of attempts for a successful attack.

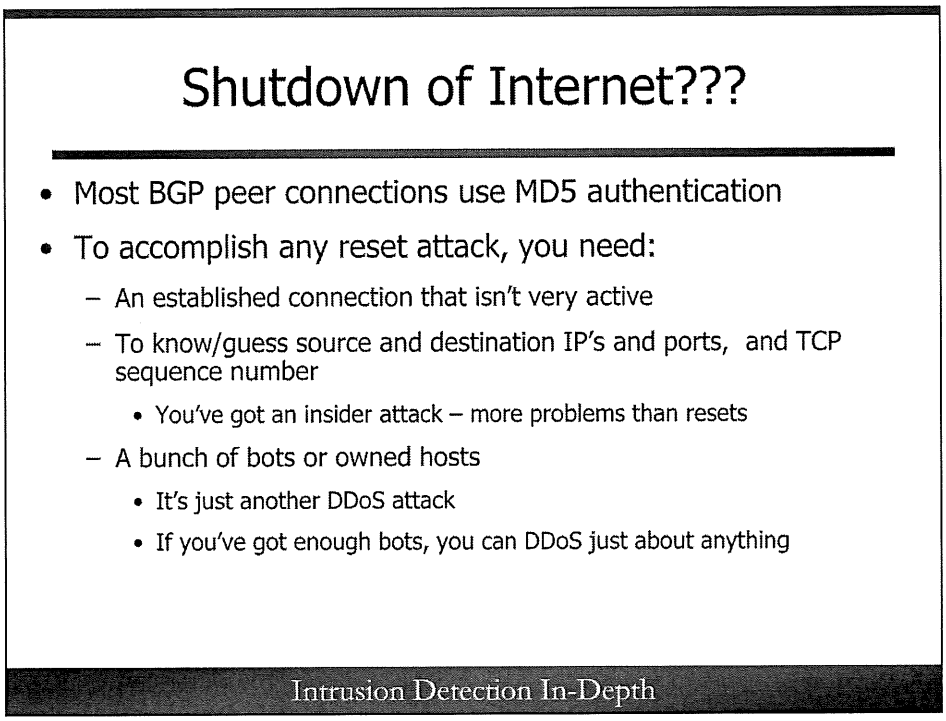


This tcpdump output shows the type of output you'd likely see if you witnessed a reset attack where a client attempts to abort the connection with a BGP server. This is a rather simplistic attack since we're starting with an absolute sequence number of 0 and incrementing it by 65,535 each time assuming that the server has a window size of 65,535.

Using the connection that we saw a couple of slides ago, we'd have to cycle to a range where the TCP sequence number of the current connection was between 1995434862 and 1995500397. The final connection has such a TCP sequence number of 1995475215.

You may also see a reset attack performed with SYN's or FIN's instead of resets. The idea of a SYN being able to accomplish a reset seems strange. You would imagine that if an established connection received a SYN packet with an appropriate sequence number in the range of the current window size (as we witnessed with the reset attack), it would be ignored. However, RFC 793 has provisions for this sort of activity and the recommended reaction is to reset the existing connection. It doesn't sound logical, but that is what the RFC recommends doing.

Also, in some testing, it was discovered that certain target operating systems, such as a Linux 2.4 kernel could be reset using a FIN with a sequence number in the window range. Usually a FIN requires a matching acknowledgement number. When this type of attack was attempted against a Windows 2000 attack, it refused the FIN and the connection remained active. However, the same attack directed against a Linux 2.4 kernel host aborted the connection.

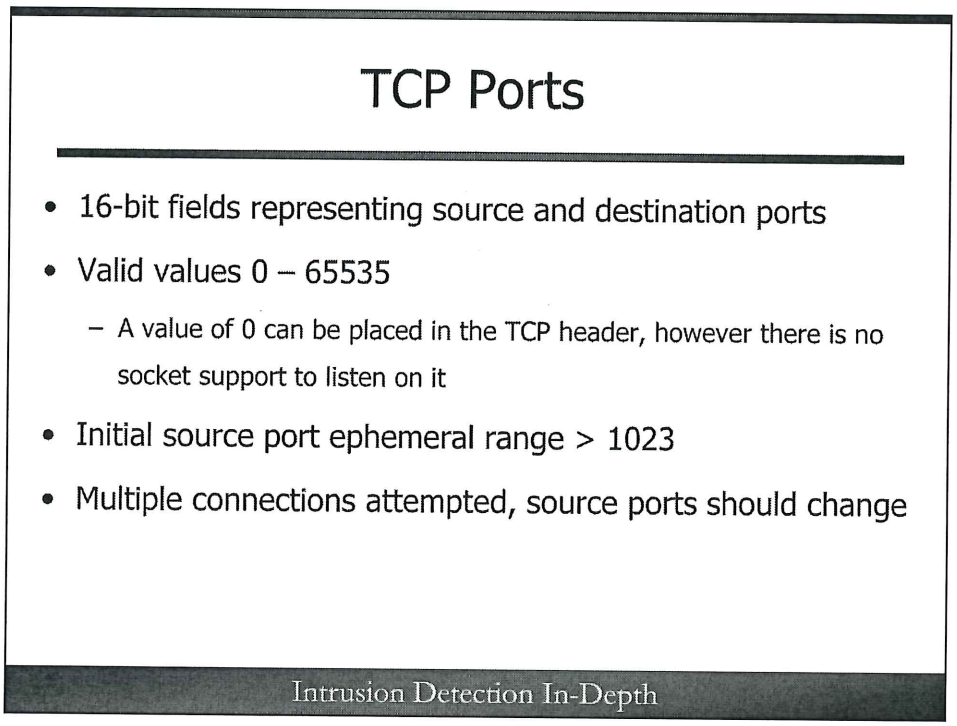


Is this attack the demise of the Internet as reported by some? Hardly! Watson took a known flaw of TCP and demonstrated that with modern bandwidth networks and a carefully selected target that is not actively exchanging data, it is possible to reset some connections. But, as far as taking down the Internet via BGP, that looks less feasible. Most, if not all, BGP backbone peers use MD5 authentication to verify that the talkers are who they say they are. Therefore, a reset attack would fail since the authentication would not exist or be valid.

That means that this attack needs a different target service than BGP to be dangerous. And, that connection has to be one that isn't too active because a moving target makes it more difficult to get the sequence numbers right. If an attacker knows all the information necessary to perform a reset attack, chances are you have an insider sniffing the traffic. If this is the case, you've got far bigger problems than annoying reset attacks.

Finally, a reset attack is more likely to succeed when distributed among many different attacking hosts. If an attacker can marshal enough resources of compromised hosts to guess source ports and sequence numbers against a target, there is a better probability that this attack can succeed. Then again, if an attacker has enough bots, any kind of denial of service can be attempted on just about anything. At that point, it may become a bandwidth issue of denying service.

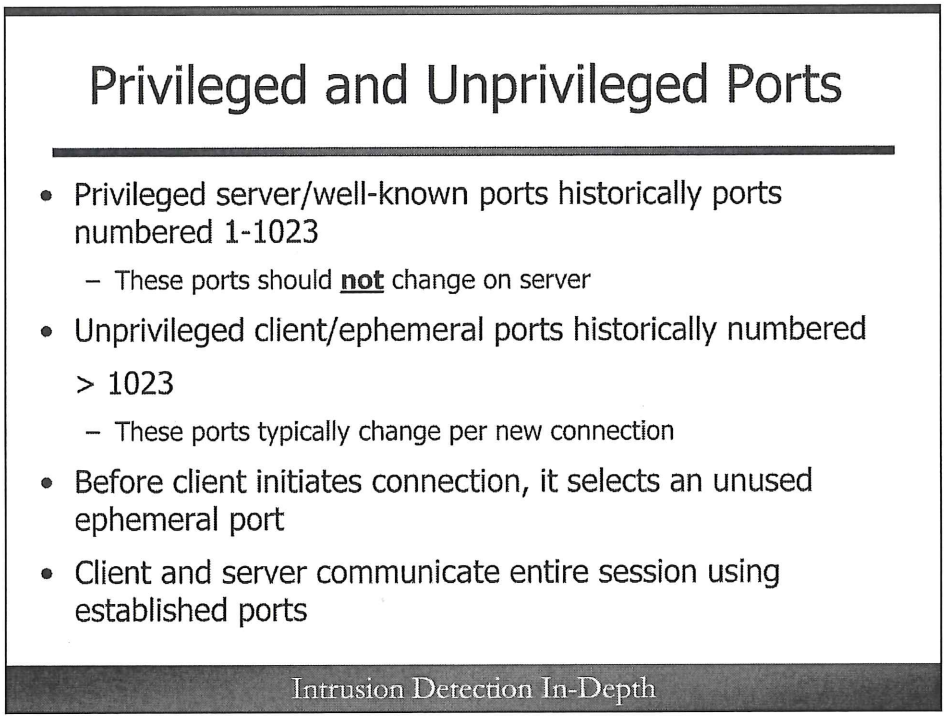
The paper provides an interesting study of an older known issue, but the reset attack is no more dangerous and perhaps far more difficult to perform successfully than other denial of service attacks. Even so, this is a thought provoking idea that teaches us more about how TCP behaves.



The port fields are two separate 16-bit fields in the TCP header, one for source and another for destination ports. The valid 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 means it can never be used for either an ephemeral or listening port.

When a source host wishes to connect to a destination host, an ephemeral port is typically selected in the range of ports greater than 1023. For each new sent connection that the source host attempts that is not a retry, a different ephemeral port should be selected. In a scan scenario, you will likely see the source port value incrementing by 1 for each new connection.

The use of port 0 should not be supported, however be aware that a SYN sent to it may respond with a RST. We'll see the possible consequences of this in an upcoming slide.

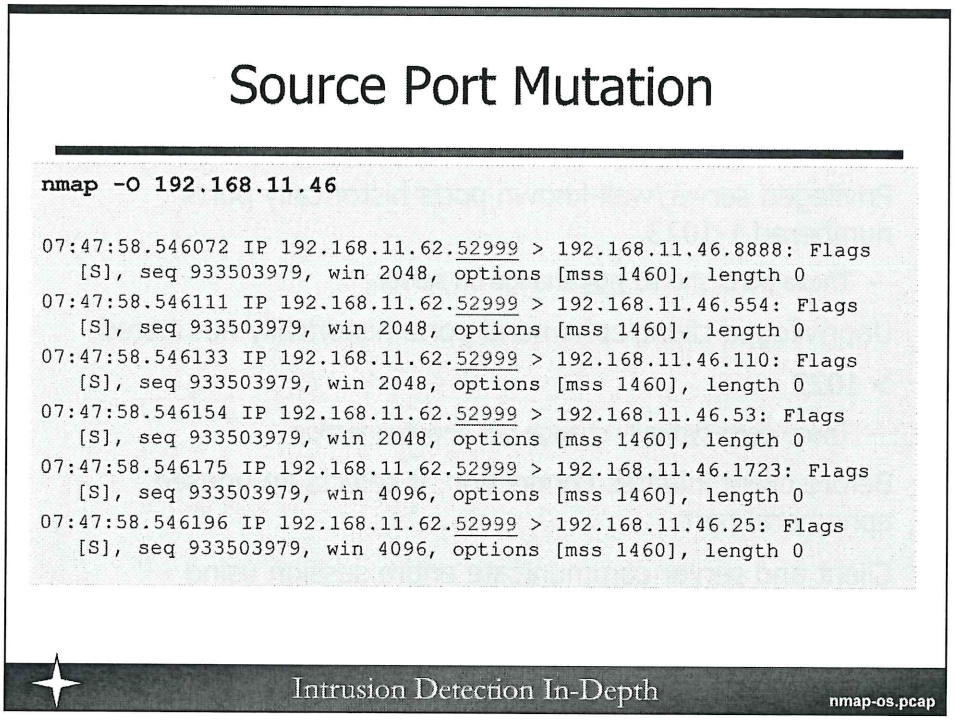


In the past, more so than today, well-known server ports generally fell in the range of 1-1023. Historically, under Unix, only processes running with root privilege could open a port below 1024. Server ports should remain constant on a given host. In other words, if you find HTTP at port 80 on a particular host one day, you should find it there the next day. You will find many of the older well-established services in this range of 1-1023 such as FTP, or SMTP on port 25. Today, many of the newer services such as alternate HTTP ports, 8000 or 8080, don't tend to conform to this original convention. This is because there are more services than numbers in this range today.

Client ports, often known as ephemeral ports, are selected only for one connection and are reused after the connection is freed. These are generally numbered greater than 1023. When a client initiates a connection to a server, an unused ephemeral port is selected. For most services, the client and server continue to exchange data on these two ports for the entirety of the session. This connection is known as a socket pair and it will be unique. That is, there will be only one connection on the Internet with this combination of source IP and source port connected to this destination IP and destination port.

There may be another user connection from another source IP to this same destination IP and destination port, but that user will have a different source IP and most likely different source port. There may even be someone from the same source IP connected to the same destination IP and port. But, this user will be given a different ephemeral port thus distinguishing it from the other connection to the same server and destination port.

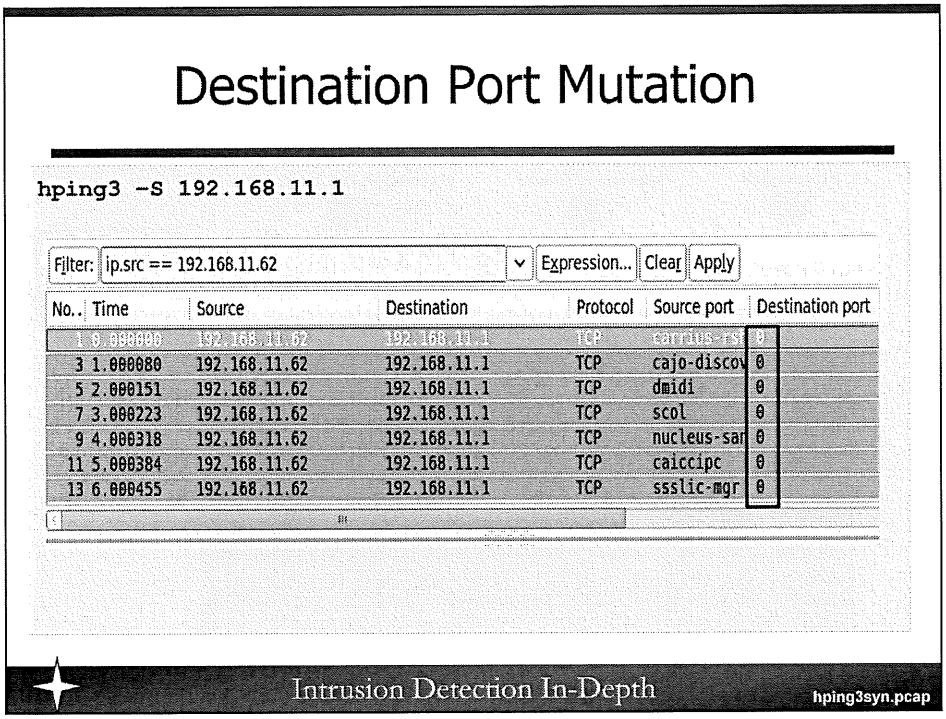
The ephemeral source port selection range of a given host may help assess the operating system since there are different selection patterns among operating systems. The Internet Assigned Numbers Authority (IANA) suggests the use of 49152-65535 for the range. More current versions of FreeBSD and Windows conform to this range while Linux uses a range of 32766-61000.



This is an excerpt of the first 6 records from running Nmap to perform a remote operating system identification scan. You'll notice that Nmap uses a default behavior of a static source port of 52999 for several TCP segments to different destination ports. If this were normal behavior, you would expect to see the source port numbers incrementing in the ephemeral port range.

To view this output, enter the command:

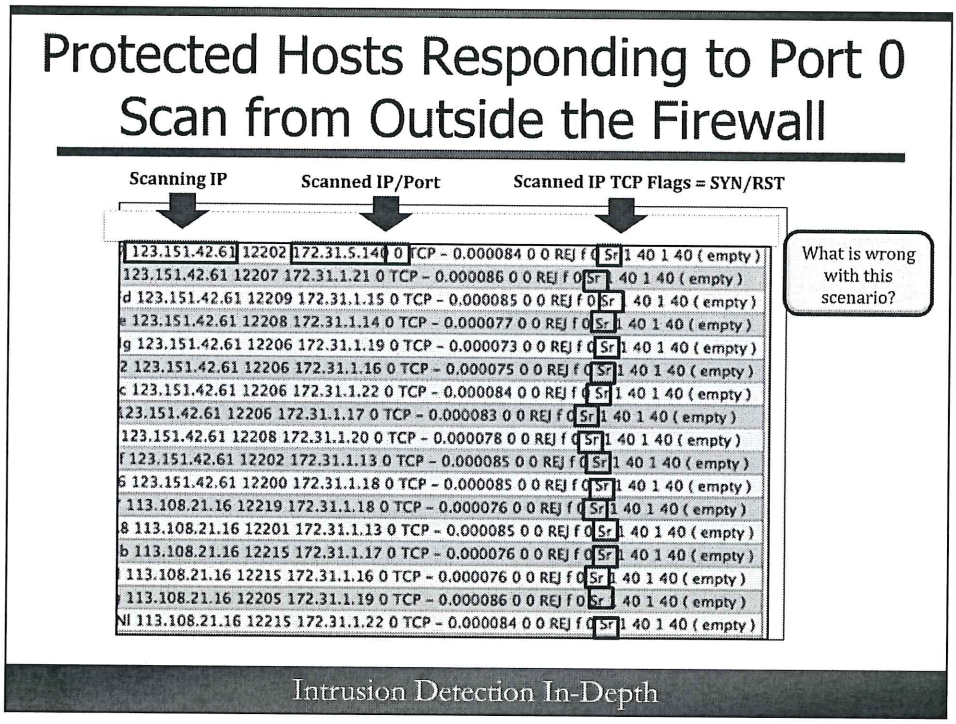
tcpdump -r nmap-os.pcap -n



Now, look at the default behavior that hping3 exhibits doing a SYN scan using the –S option. It uses destination port 0 as its default target. The intent of this type of scan obviously is not to find a listening port. This type of scan would be used to elicit a RESET response to see if a host is alive since a host should not be listening on port 0.

If you would like to see this output, enter the command:

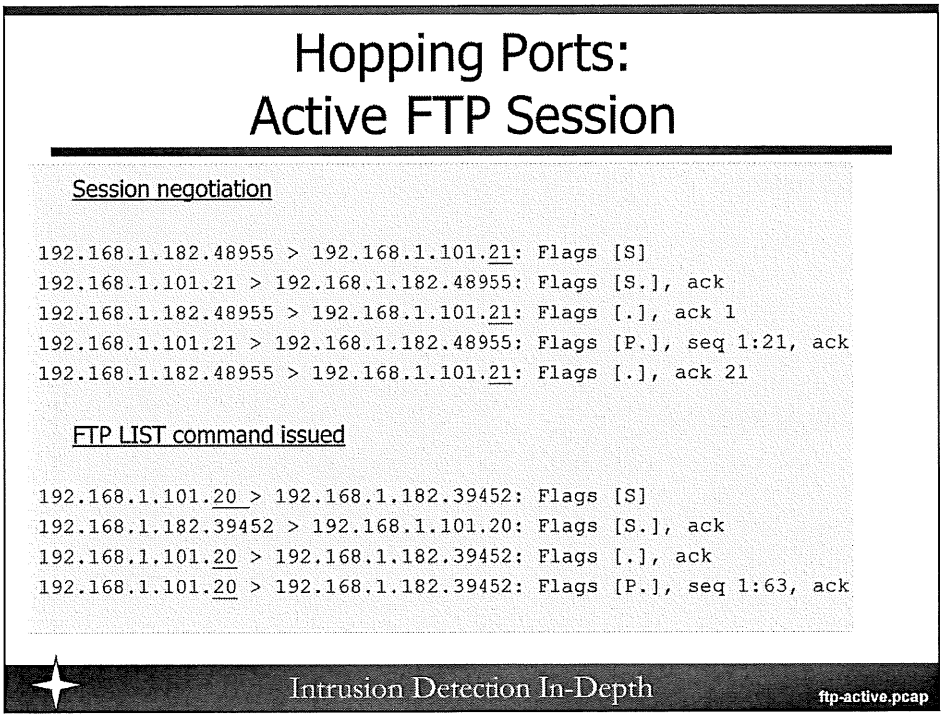
wireshark hping3syn.pcap



Take a look at some partial Bro output from a scan of some internal 172.31.0.0/16 network hosts. There are two scanning hosts, 123.151.42.61 and 113.108.21.16, observed that are sending SYN's to TCP port 0 on various 172.31.0.0/16 hosts. The internal hosts respond with RST. The "Sr" in Bro parlance indicates that the source IP sent a SYN designated by an uppercase "S" and the destination IP returned a RST designated by the lower case "r". This is what you would expect as a response from a normal server to respond to an attempt to establish a session with port 0.

But, the responding hosts are inside a firewall that should have blocked inbound TCP traffic to port 0. What does that indicate? The administrator who configured the firewall access control rules failed to consider and block TCP port 0. True, the port is closed on all the scanned hosts, but this informs the scanner that the hosts are live and potential targets.

This brings up a pertinent point that your IDS is not just for intrusion detection, it can also be used for audit functions. This scan response highlights to you that there is an issue with the firewall access control rules that need to be corrected to block the inbound TCP port 0 traffic.



Let's look at an unusual, but normal, hopping of TCP ports. The expected behavior of TCP that we have witnessed so far is to establish the 2 ports used by the client and server during the three-way handshake. The client usually selects an ephemeral port greater than 1023 and the server listens on a well-known port. Throughout the remainder of the established TCP session, the client and server talk on the established ports only.

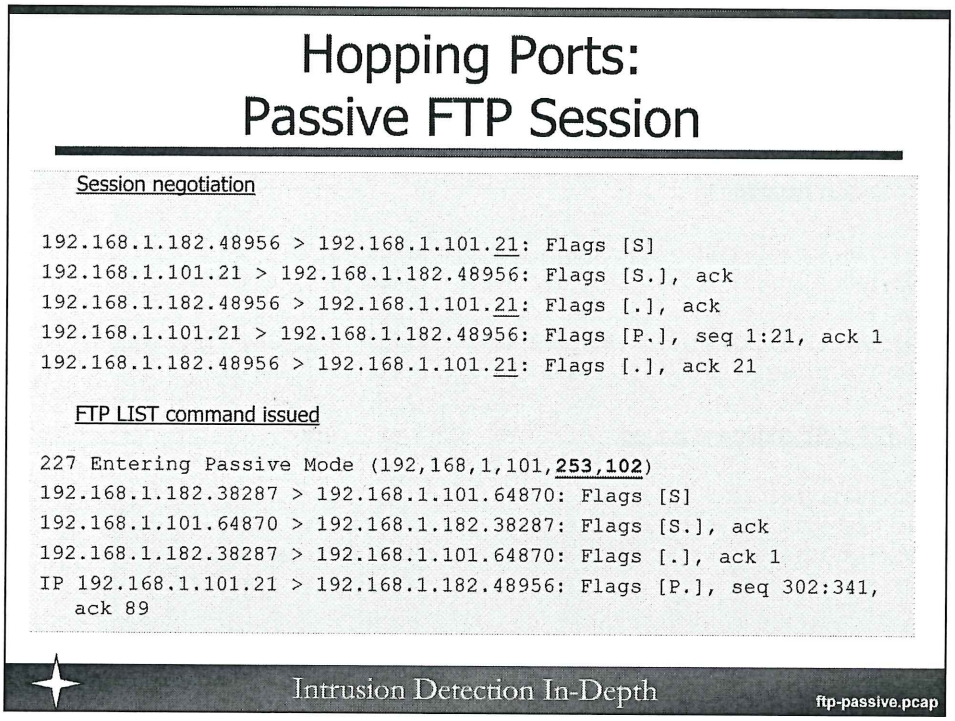
FTP port 21 for Commands port 20 for Data

FTP is different in that it communicates using 2 different server ports. There are 2 different types of FTP connections – active and passive. We'll describe active FTP first. The initial port connection on the server for either passive or active is port 21. It is known as the standard FTP command port. For active FTP, the second port connection is source port 20 from the server that is used for FTP data passed between the client and the server. Port 21 is used to issue FTP commands such as those used to retrieve or store a file. The FTP data port is used to exchange a file between the two hosts or other exchanges that require data to be sent – such as a listing of file directories.

In the above example, assume that the client 192.168.1.182 is in the protected network and 192.168.1.101 is an external FTP server where there is a filtering device between them. We see that the FTP connection is established between 192.168.1.182 and 192.168.1.101 using ephemeral port 48955 and server port 21. The three-way handshake is completed and some data, typically a welcome message, is passed between the two. Next, the FTP LIST command is issued from the client to list the remote files on the server. A new connection is established from source port 20 of the server to a new ephemeral port 39452 on the client. After this new three-way handshake is completed, the 192.168.1.101 host can send a list of the files to 192.168.1.182 using this established connection.

To view this output, enter the command:

tcpdump -r ftp-active.pcap -nt



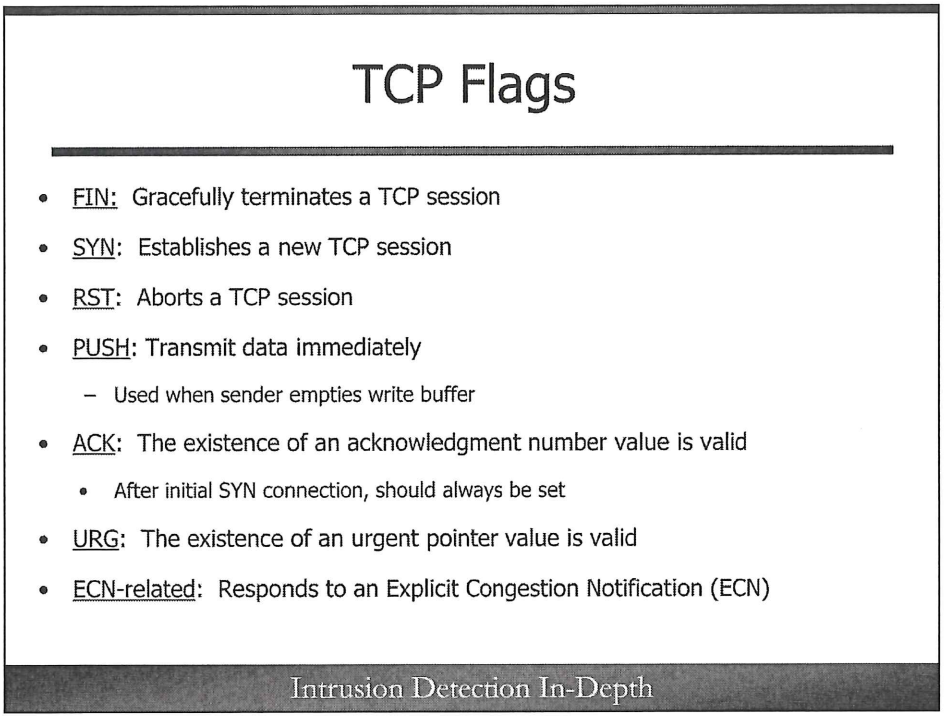
Passive FTP differs from active FTP. The problem with active FTP is that the server has to initiate a data connection from source port 20 outside the network to a high-numbered port on the client inside the network. This could cause problems for filtering devices such as firewalls that are required to open up access to high-number ports to support this.

Passive FTP avoids this problem by having the client open up the data port to the server. In the above exchange, we see the same initial FTP connection to negotiate the session commands and allow the user to log in using server port 21. But, after the user logs in and issues a LIST command, a new data channel connection must be established. This time, the server responds saying that it is entering passive mode and gives some kind of cryptic command (192,168,1,101,253,102).

The cryptic command tells the client to make the connection to destination IP 192.168.1.101 and further tells it the port that it will be listening on is 253,102. The 253,102 is the port number that is a 2-byte field. The high-order byte has a value of 253 and the low-order byte has a value of 102. If you were to represent these two bytes in hex, the value would be 0xfd66 – a decimal value of 64870 as seen as the server port number above.

Then, the client establishes a connection to server port 64870. Because this is from the internal network outbound, it avoids the active FTP problems of attempting to establish a new connection from outside the network. The way to remember which is "active" versus which is "passive" FTP is to think about the server's role in establishing the data connection. In active FTP, the server actively connects to the client, but in passive FTP the client connects to the server, making the server a passive participant.

To view this output, enter the command:
tcpdump -r ftp-passive.pcap -nt



TCP flags or code bits represent a bit or multiple bit settings in the 13th byte offset of the TCP header. 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 normally used to gracefully terminate a session – for instance, by doing an exit in a session. Because TCP is full duplex, both sides of the connection have to gracefully close by sending individual FINs in each direction and the receiving host acknowledging those FINs.

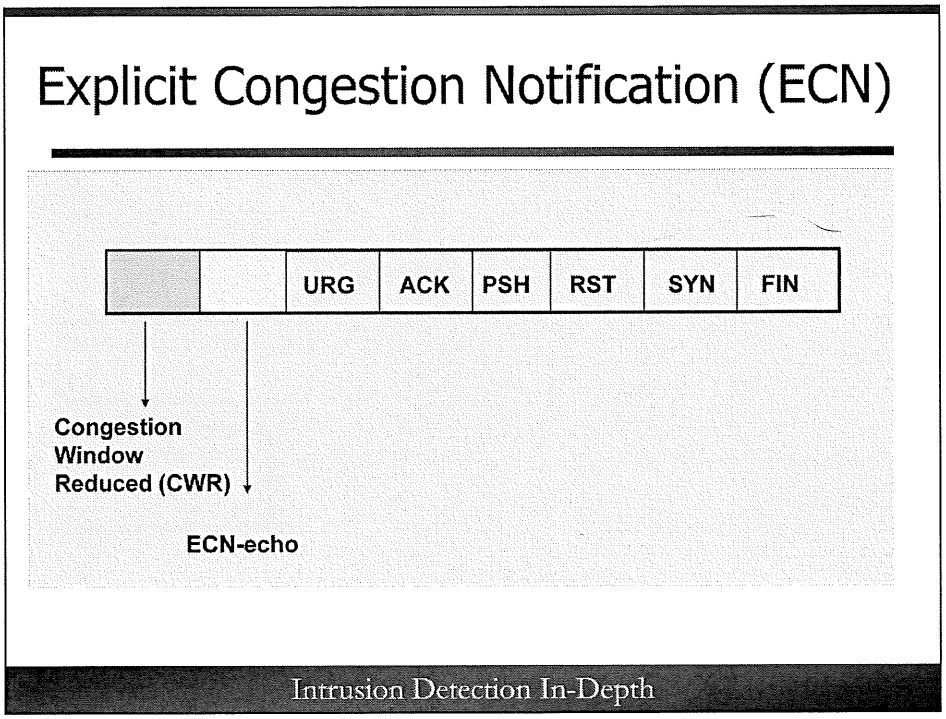
The SYN (synchronize) flag is set only when a host wants to establish a new session with another host. Because data must flow both ways on a TCP session, it is full-duplex, and each host involved in the session must establish its own connection to the remote host. The SYN is set on the first two segments 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. It is expected that there will be no response to a RST.

A sender has a write buffer where it stores data that is to be transmitted to the receiver. Multiple segments may be required to send the data depending on the size of the payload. The PUSH flag tells the sender to empty its write buffer and informs the receiver to push the received data to TCP and the listening application. Suppose the sender has multiple segments to send. The last segment only has the PUSH flag set (the ACK flag also will be set); all previous segments from the same write buffer will have the ACK flag alone set. In essence, the PUSH segment denotes the last segment belonging to the current write buffer. All these segments should contain data.

The ACK flag quite simply stipulates that the existence of a value found in the acknowledgment number field of the TCP header is valid. For instance, it is possible for a TCP/IP stack to set a non-zero acknowledgment number value in the SYN segment. The associated acknowledgment number value will be ignored because the ACK flag is not set. The ACK flag will always be set in normal traffic in every TCP segment after the completion of the three-way handshake.

The URGENT flag is not used often in modern operating systems, well at least for legitimate purposes. Like the ACK flag it signals the existence of another value, the urgent pointer, in the TCP header is valid. The URG flag and associated urgent pointer value signal that there is some urgent (or sometimes called out-of-band) data that must be sent before other data is sent. The urgent pointer value denotes the byte offset into the payload where the urgent data is found. The URG flag and pointer were used for an interrupted session (user presses CTRL-C).

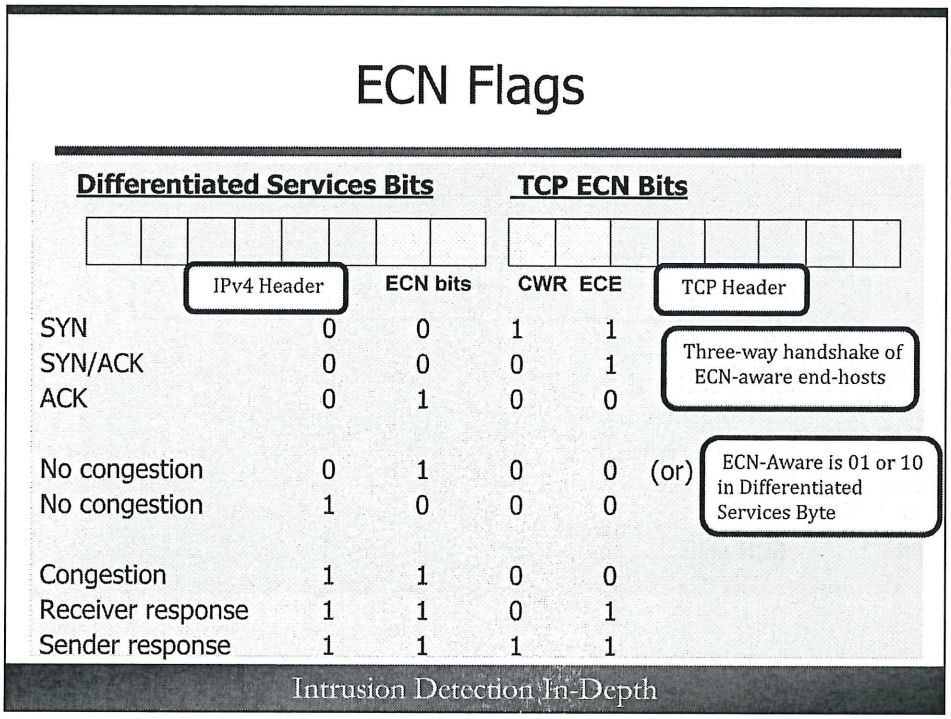
There was a time when the two high-order bits of the TCP flag byte were reserved. However, Explicit Congestion Notification (ECN) is a scheme that relies on the use of these two high-order bits. ECN is a mechanism used in TCP to reduce detected congestion in a network by having hosts contributing to the congestion reduce the amount of traffic sent. The high-order bit is known as the Congestion Window Reduced bit. It signals that the sender will send less data. The bit to the right of that is the ECN-echo bit and it is set as the result of congestion experienced. If you would like to read more about ECN, reference RFC 3168. ECN is discussed in more detail in subsequent slides.



Remember back when we were discussing the old IPv4 header type of service byte – now known as the differentiated services byte, renamed as traffic class in IPv6, we introduced a future purpose for the two low-order bits known as Explicit Congestion Notification (ECN)? The intent was for a router to be able to notify a sender that there was congestion in the network and to reduce its sending rate.

Well, how exactly does that occur? Currently as discussed in the ECN RFC 3168, the only transport capable of reacting to that congestion notification is TCP. So, TCP has to be prepared to deal with this. The RFC offers guidance to use the two high-order bits of the TCP flag byte as fields for ECN. The bit to the right of the high-order bit will be known as the ECN-echo bit. This 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. This assumes that both end-points of the TCP conversation are ECN-capable and that is determined during the three-way handshake.

If TCP sets the ECN-echo bit, the purpose is to inform the other side of the conversation to reduce the rate at which 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 half. Once it reacts in this manner, it will turn 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 byte flag.

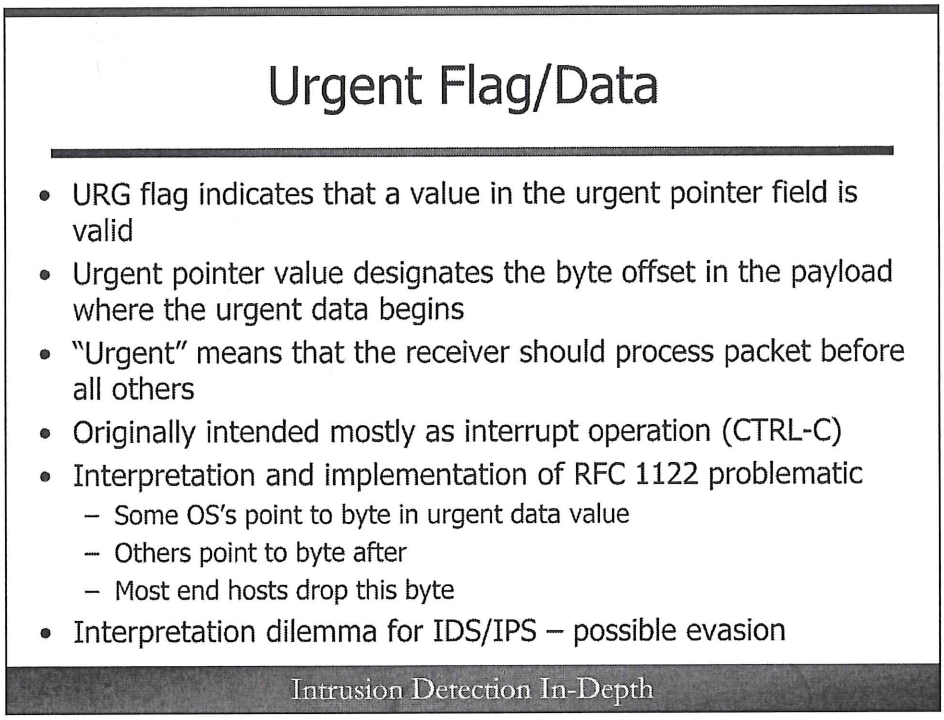


Let's take a look at the settings of the ECN flag bits in both the differentiated services byte in an IPv4 header and the TCP flag byte when both end-hosts are ECN-aware. The first phase of ECN is informing that both end-hosts are ECN-aware. Otherwise, if either or both are not, it makes no sense for a router to mark a packet congested if no possible remediation can occur.

When an ECN-aware host initiates a SYN, it will set 0's in the two low-order ECN bits found in the differentiated services byte. Yet, it will mark the two-high order ECN bits of the TCP flag byte as set. This will inform the receiving host that the sender is ECN-aware. If the receiver is ECN-aware, it will set the SYN and ACK flags as expected, but it will set the ECE bit only. When the original sending client receives these settings, it will know that the server is ECN-aware and will clear the ECN flag bytes using the differentiated services bits as the means to inform any router that the end-hosts are ECN-aware. It does so by using one of two pairs of bit settings in the ECN bits of the differentiated services byte. These bit values will be either 01 or 10. A value of 00 in these bits either means that the end-hosts are not ECN-aware or performing the ECN negotiation as we just did. You may wonder why the ECN-aware bits moved from the TCP flag byte to the differentiated services byte. First, we have to clear the flag byte ECN bits for the purposes of reacting to congestion and also we move to the network or IP layer for routers and the differentiated services byte is located in the IP header.

If congestion is detected and marked by the router, the ECN bits in the differentiated services byte will both become 1. And, the receiving host will have to respond by setting the ECE bit to 1. The original sender will get this bit setting and inform the other host that it is responding to the congestion by assigning a value of 1 to the CWR bit.

This is a lot of complication for a feature that is rarely used. Someone thought that this would be a good idea, yet it has never really caught on because routers and hosts have to be ECN-aware and deal with any congestion. Most modern operating systems are able to deal with ECN even though it is infrequently used in practice.



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 will be used. This segment has data that the receiving host should process before all others that may be queued in its buffer. The urgent data field in the TCP header is 16 bits and it is supposed to point to the byte offset where the urgent data begins.

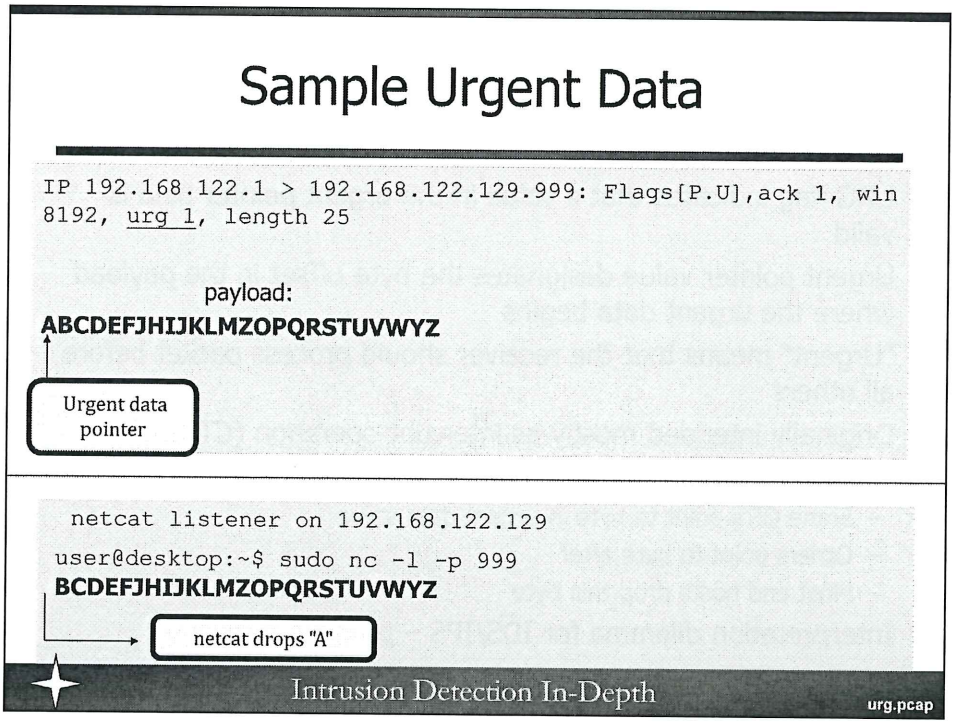
Some implementations have interpreted the location of the urgent pointer to be the exact byte value specified in the urgent data field, while others have interpreted it as the byte following this value. So, what's the fuss and why do we care as analysts which interpretation a host adopts? Many receiving 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 as 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.

An excellent discussion on evasions using the urgent pointer is found at:

http://phrack.org/issues/57/3.html

Look for the article named:

NIDS Evasion Method named "SeolMa"



In the above output, you see the host 192.168.122.1 sending host 192.168.122.129 a segment with 25 bytes of the poorly entered letters of the alphabet. The urgent flag is set and the urgent value is 1. This means that the above payload byte at offset 1 (counting starts at 1) is considered to be the start of the urgent data.

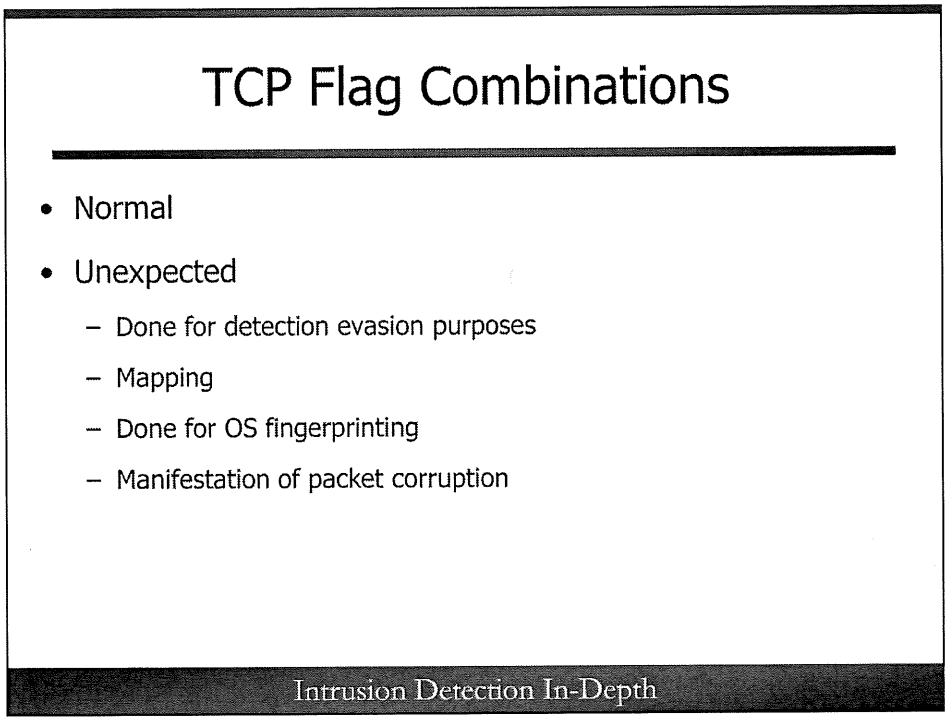
A netcat listener on receiving host 192.168.122.129 displays the data it receives. As you can see, it looks like the TCP stack on 192.168.122.129 has dropped the byte found at offset 1 – the "A". Hypothetically, let's say we had a rule that fires when the payload of "ABCDE" is found in the packet. And, let's say we craft a packet that has a payload of "AXBCDE" and an urgent value of 2 and the urgent flag set. Most likely, the IDS/IPS will see "AXBCDE", but the listening application will see "ABCDE". This would be an evasion if "ABCDE" is considered to be malicious.

This whole issue gets more complicated because even if the TCP/IP stack does not drop the urgent data, a receiving host application may. In this instance, it is not possible to have some kind of target-based awareness in the IDS/IPS because it can't possibly know every application and associated urgent data behavior.

As if that is not bad enough, you have interesting cases of whether or not a receiving application will drop 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 very large number? Does the IDS/IPS set aside resources to track this or perhaps an attack where all segments have large values in the urgent pointer?

The point is that this is such a complex issue that chances are your IDS/IPS can be tricked into missing some kind of attack.

To view the output, enter the command:
tcpdump -r urg.pcap -nt

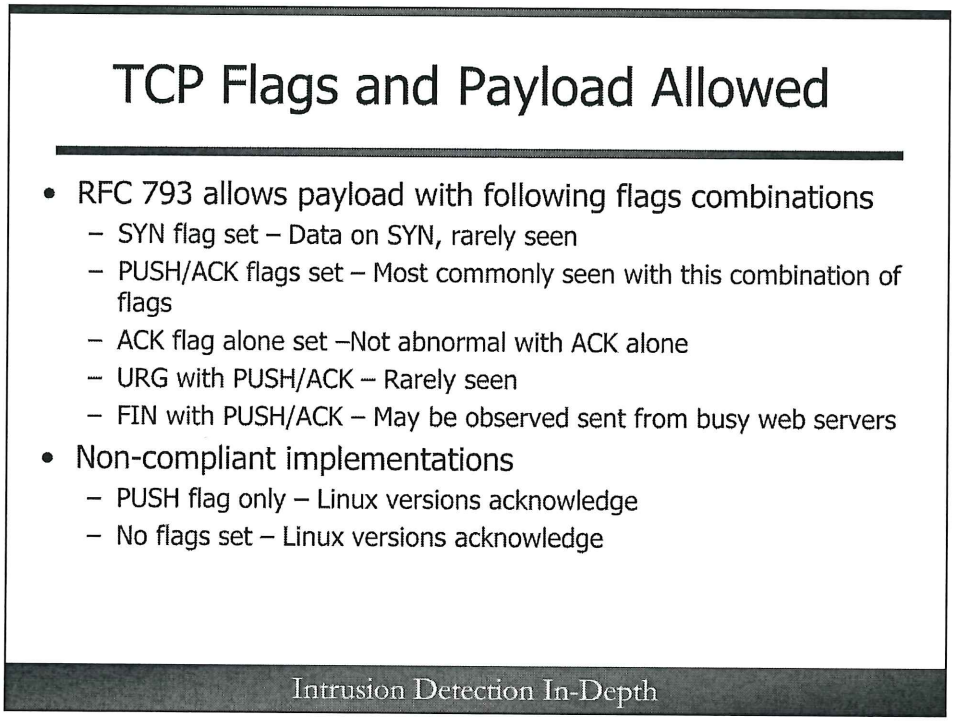


As you have witnessed, the TCP flags have many different valid normal combinations. And, there are many different invalid combinations that are used for different purposes.

Early on in the evolution of IDS's, many would examine traffic for initial SYN attempts only. Attackers realized this and would send a SYN/FIN combination that might elicit a response from a host. Different operating system TCP/IP stacks respond differently to mutant flag settings so this can be used to attempt to fingerprint the operating system. If an attacker were to map hosts in a remote network using a SYN scan, it might be detected since it is such a common scan. Yet, if the attacker used a strange combination of flags as a scanning method, it might not be as apparent. The receiving host would likely issue a RST since there is no established connection, permitting the attacker to find live hosts.

Nmap may use strange combinations of TCP flags in an attempt to elicit responses that help identify an operating system since there may be unique responses by different operating systems.

Finally, just because you see mutant TCP flag combinations, it is not necessarily an indication of malicious behavior. Packets can and do get corrupted and it is possible for TCP flags to be unnaturally set after some kind of corruption in the TCP portion of the packet. This snapshot of the corrupted packet can occur before the packet is received and dropped by the end host since it will have an invalid TCP checksum.

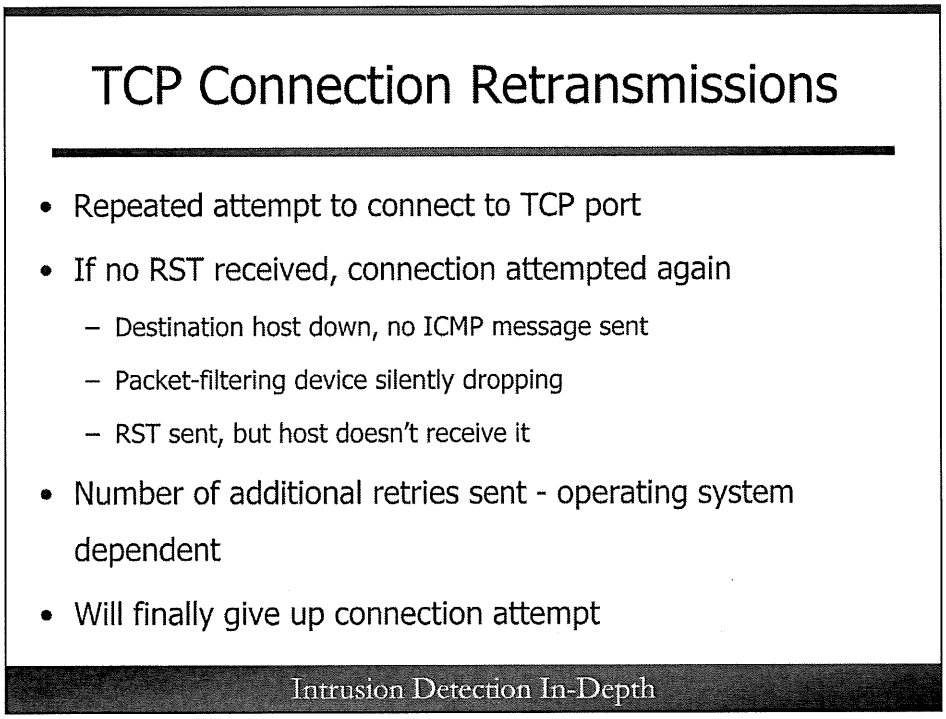


An IDS/IPS must begin to analyze payload only after a TCP session has been established. Years ago, attacks such as Stick and Snot attempted to cause problems for an IDS that did not ensure that a TCP session had been established before analyzing payload. The result was that the IDS fired false positives to the point of distraction for the analyst. Nowadays, any reputable IDS/IPS keeps track of TCP session state as it analyzes payload data.

RFC 793 offers a provision for a host to send data on the SYN packet. Most operating systems will not acknowledge this data even though the RFC offers guidance that it is acceptable. Once, the TCP session has been esfablished, 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 since 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. 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.

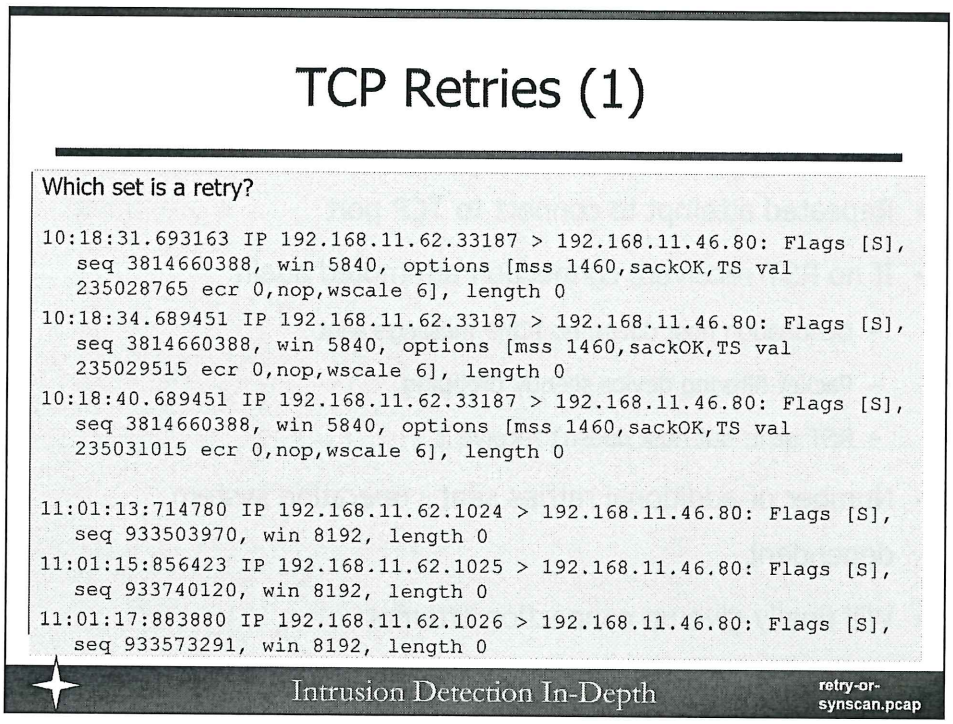
Just because RFC 793 supplied the guidelines for proper TCP flag settings does not mean that all TCP/IP stack implementations adhere to this guidance. For instance, more current Linux versions allow data to be sent where the PUSH flag alone is set or stranger yet – when no TCP flags are set. No other well-known operating system acknowledges such data on segments with these TCP flag configurations.

A good IDS/IPS will follow the RFC guidelines, but also attempt to understand the aberrations such as Linux when it does payload analysis.



What if a TCP connection is attempted, yet the host attempting the connection doesn't receive a response from the destination host? A destination host might not respond because it may not be up or may not exist. A router may attempt to deliver an ICMP message about the destination host being unreachable, but if the router has been silenced from delivering unreachable messages, the sending host will never know that there is a problem. A destination host 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 destination host responds positively (SYN/ACK) or negatively (RST/ACK), yet for some reason the sending host doesn't receive these replies.

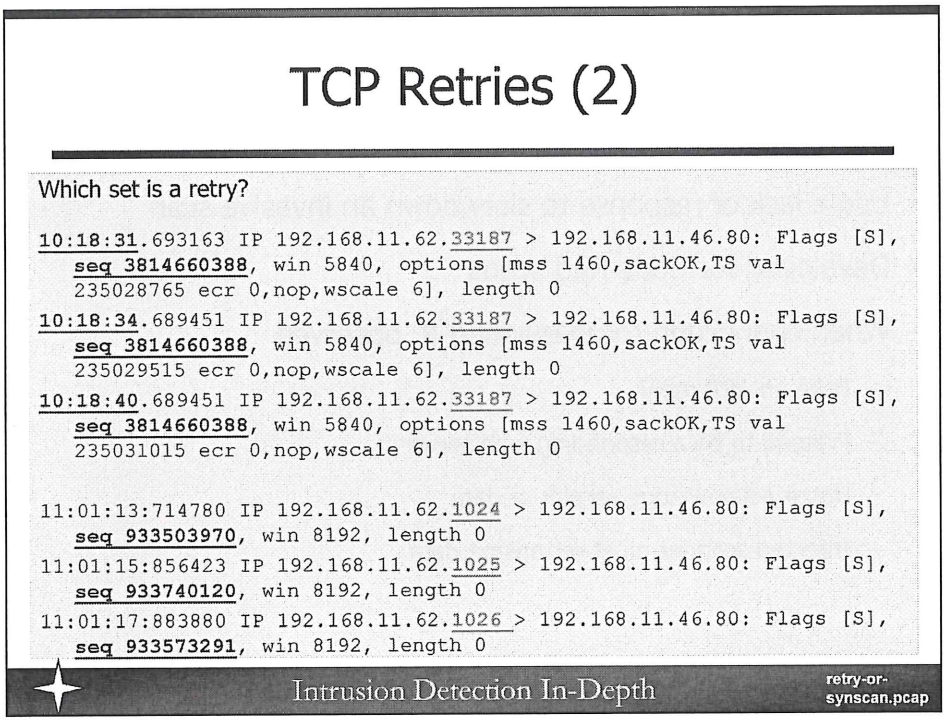
Additional attempts or retransmissions will be made to contact the host in situations like this. The number of retransmissions and the time intervals between attempts vary by operating system. Eventually, the sending host will cease sending the connection attempts.



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 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?

- To view the output, enter the command:

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

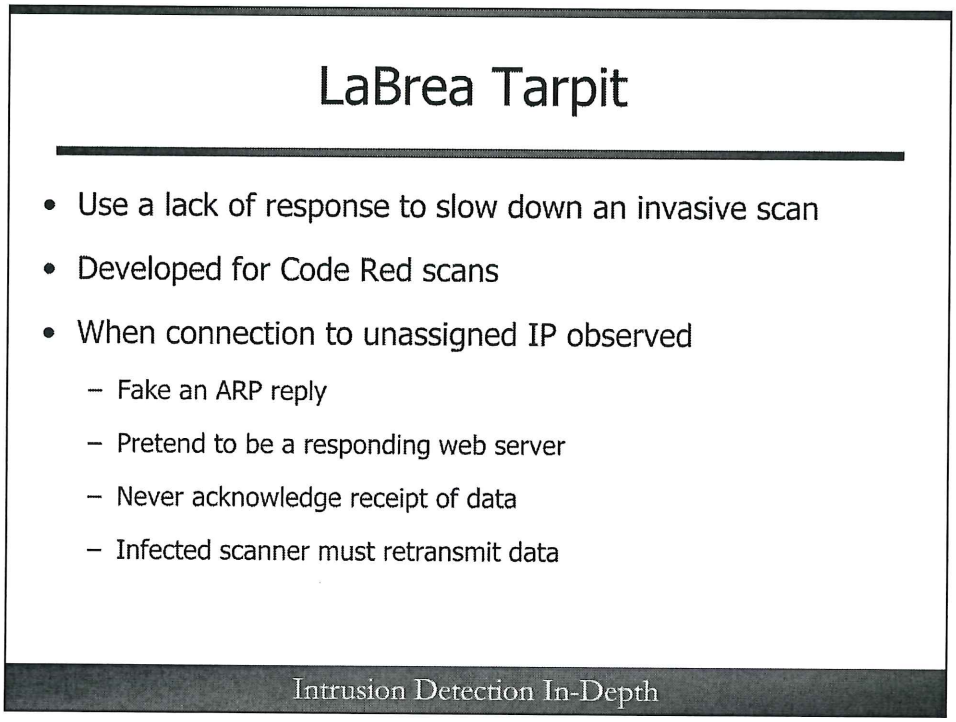


The first set represents the retry and the second a SYN scan. Both have the same source and destination IP addresses and destination port 80. The retry is different in several ways. Many operating systems use something called a back off timer that dictates the number of seconds between each retry. The operating system above doubles the retry for each successive try. There is a 3 second difference between the first and second retry and a six 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 seen above. The packet essentially remains the same for each retry.

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 command:

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

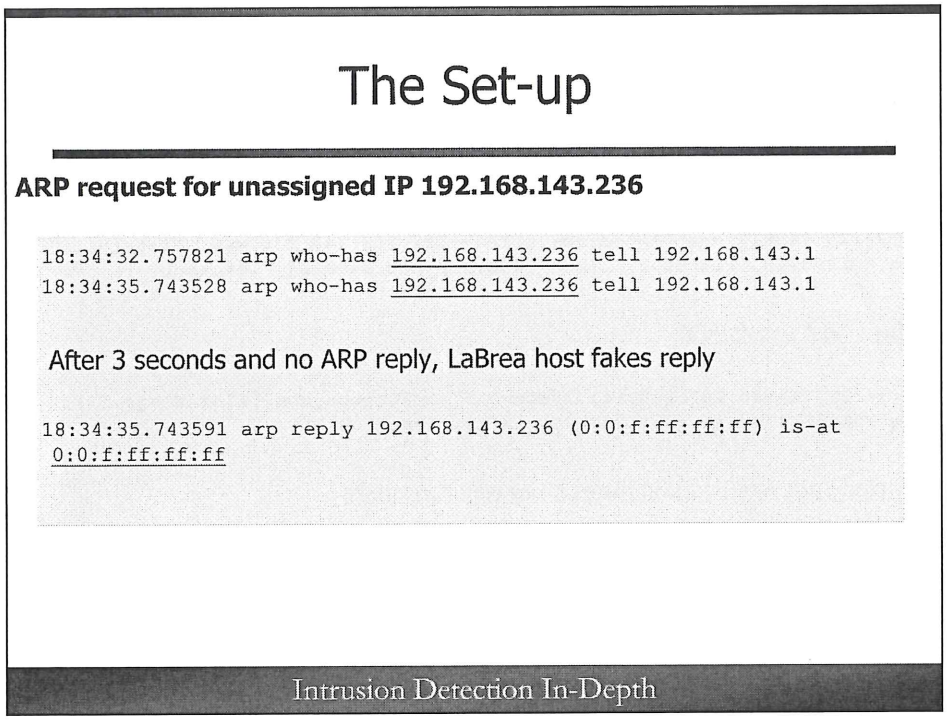


A very clever defender against the Code Red worm scans for web servers, Tom Liston, wrote a program that "tarpits" scanners looking for unassigned IP numbers. Code Red was a worm that debuted in 2001 to exploit a vulnerability in the Microsoft IIS web server. Typically, when you see activity to many unassigned IP addresses, it may mean someone is scanning hosts on your network. Tom named his code LaBrea after the La Brea Tar Pit*. While developed for Code Red, this technique could be used for any nuisance scans.

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

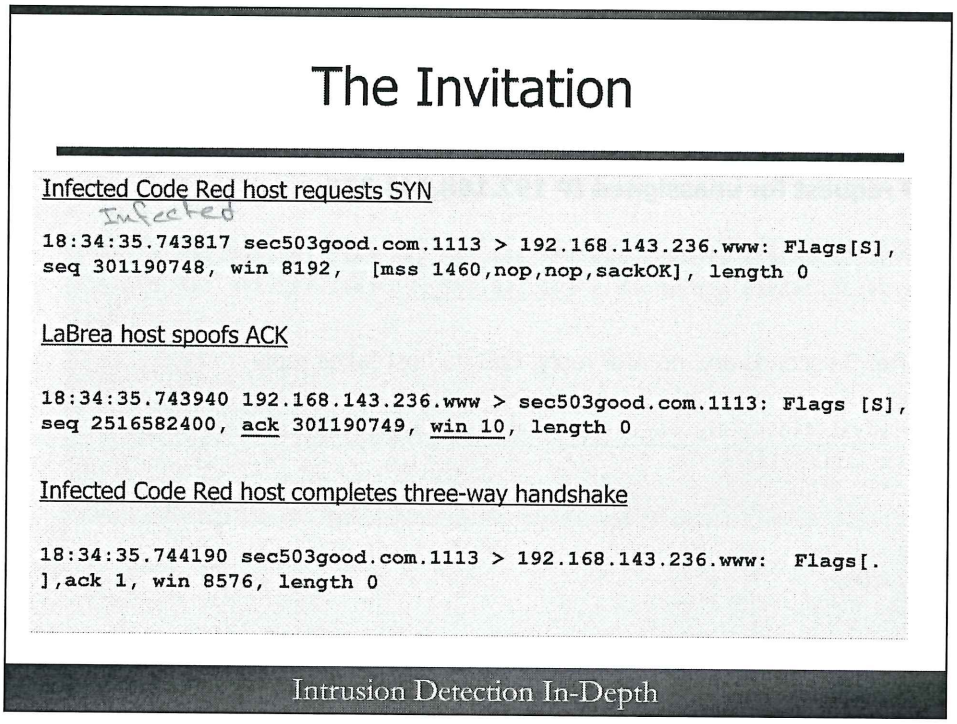
LaBrea is available at http://labrea.sourceforge.net/labrea-info.html

*La Brea Tar Pit was a sticky thick oil seep that formed over 2.5 million years ago in what is now Los Angeles's Hancock Park. Animals that sought water from or attempted to cross the tarpit became trapped and died.



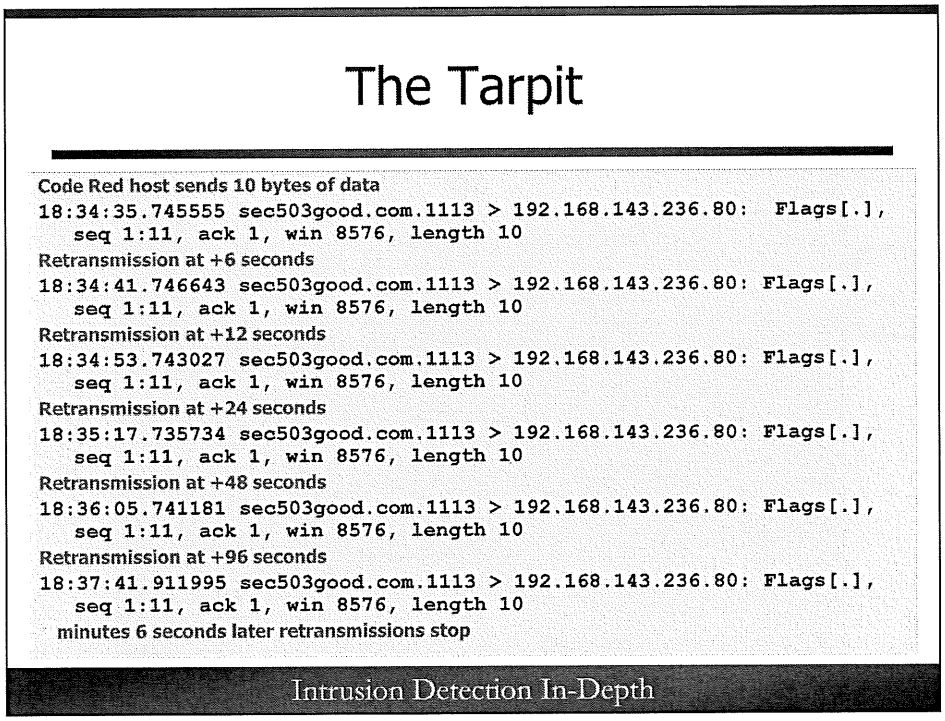
LaBrea first looks for ARP requests on the local network. These will usually come from the local routing device. If it sees no ARP reply after 3 seconds (the default wait time, however it can be changed by a LaBrea command line option), LaBrea will fake an ARP reply. In this case, we see an ARP request for host 192.168.143.236 from the local router, 192.168.143.1. This is an unassigned IP number. No ARP reply is seen and another ARP request is generated 3 seconds after the initial one.

After 3 seconds, the LaBrea host fakes an ARP reply and tells 192.168.143.1 that the MAC address for 192.168.143.236 is a bogus 0:0:f:ff:ff:ff. Neither the 192.168.143.236 address nor the MAC address is an actual IP address in the network. This is a way to allow the routing device to respond to the scanner without generating an ICMP unreachable error. Now, the LaBrea host will look for any traffic going across the network destined for the bogus MAC address.



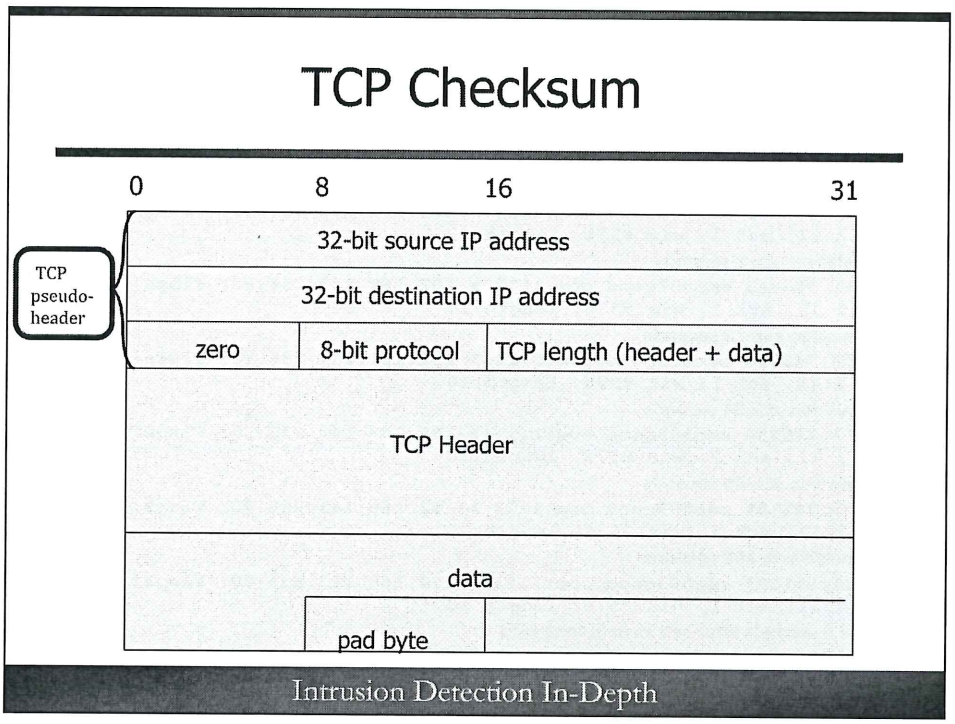
Once the bogus IP to MAC address pairing is generated by LaBrea, the scanning host most likely will attempt a connection. If this is a TCP connection, regardless of the port, LaBrea will simulate a listening host.

In the above slide, you see the sec503good.com, a host infected with Code Red, attempts a SYN connection to the unassigned destination IP address 192.168.143.236 destination port 80. LaBrea then generates a response to this connection with a SYN/ACK from the non-existent IP address 192.168.143.236. And, as expected, the sec503good.com host completes the three-way handshake. The connection is now "established". One of the other enhancements included in the bogus response is a TCP window size of 10. As you will learn in an upcoming slide, the TCP window size is the receive buffer for the host for this particular connection. A window size of 10 is much smaller than a normal window size.



Next, the sec503good.com host attempts to send 10 bytes of data to fill the receive buffer of the bogus web server 192.168.143.236. There is no PUSH flag set as you are used to seeing because the PUSH flag is usually only set when the sending host empties its sending buffer. But, because sec503good.com's send buffer is greater than 10 bytes, the only flag you see is the ACK flag acknowledging receipt of the bogus initial SYN connection from 192.168.143.236.

Now, here comes the tarpit. There is no acknowledgement of the data sent by sec503good.com. So, sec503good.com must retransmit the data. The retransmission timer for this particular host has an exponential back off where it doubles the time between retries. Five retries and three minutes and six seconds after the initial attempt to send data, the sec503good.com host gives up. But it has expended resources for sec503good.data and has been delayed in its scanning for this duration. If the scanning host waits for the response from the scanned host before continuing other scans, then it has been slowed down in its efforts. This will be more effective if the scanning host is tarpitted over and over again for all unassigned IP's on this network. Even if the Code Red code is multi-threaded and able to scan multiple hosts concurrently, it is slowed in its progress when it encounters any host that runs LaBrea.



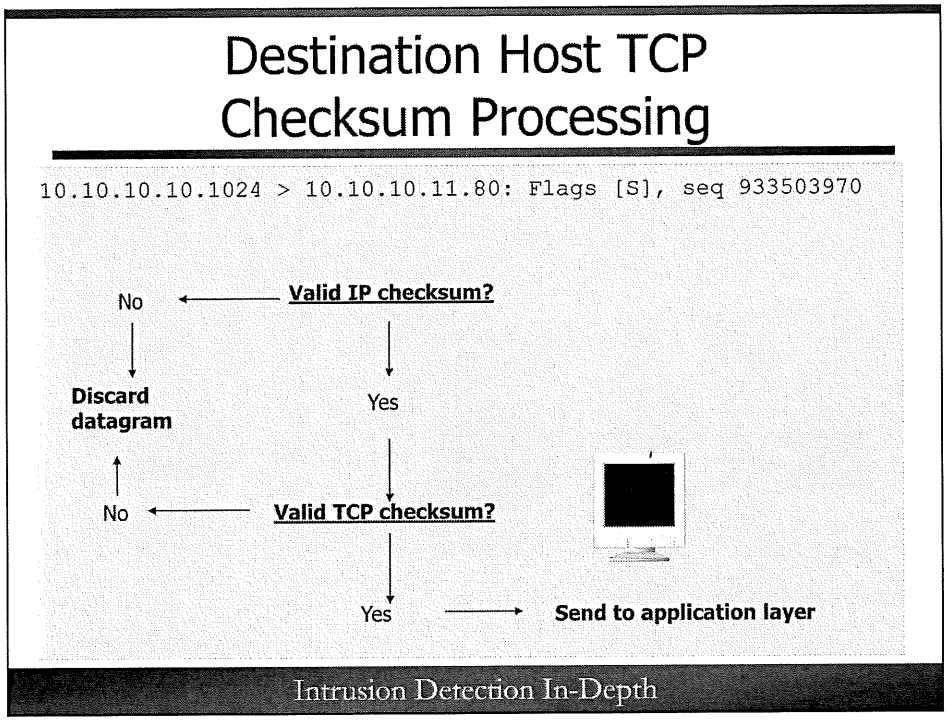
As mentioned previously, the embedded protocols have checksums as well. These cover the embedded protocol header and data. Unlike the IPv4 IP checksum, these are end-to-end checksums calculated by the source and validated by the destination host only. The TCP checksum has been chosen to show what the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

UDP & TCP Check Sum will Change if IP head Changes

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 above slide - the source and destination IP's, the 8-bit protocol found in the IP header and the embedded protocol header plus data length. The zero-pad field found in the 8th byte is used to pad the 8-bit protocol field to 16 bits since checksums are performed on 16-bit blocks of data.

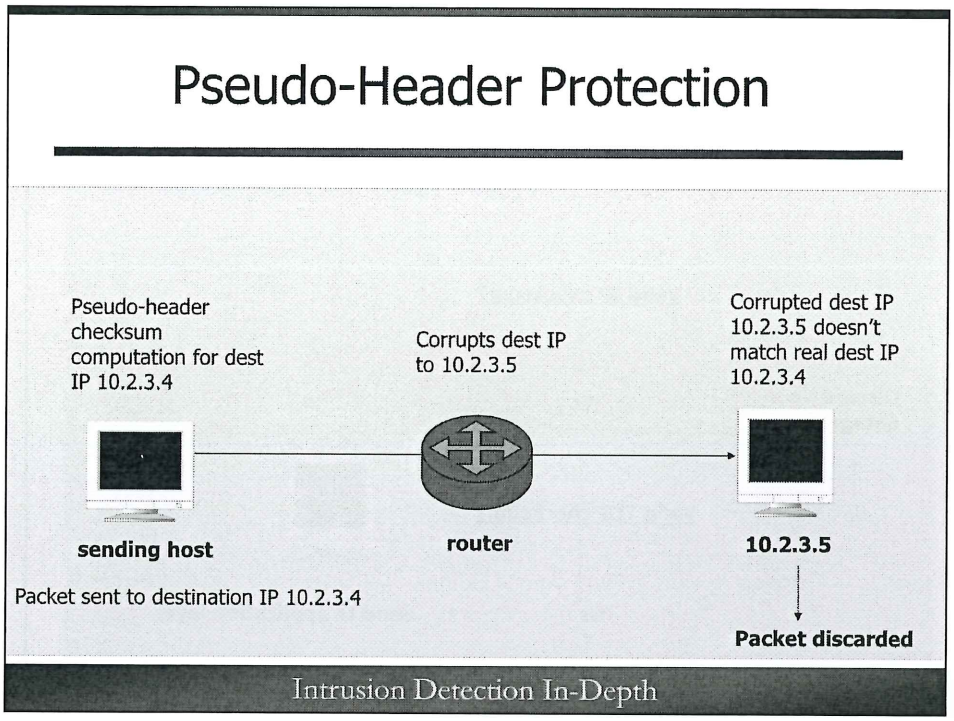
Why is the pseudo-header necessary? This is a double check that is used by the receiving host to validate that the IP layer has not accidentally accepted a datagram destined for another host or that IP has not accidentally tried to give UDP or TCP a datagram that is for another protocol. If there is some errant 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 if you are doing some kind of packet manipulation, checksums may come back to haunt you – especially the concept of the pseudo-header information. Not long ago, I volunteered to obfuscate the IP addresses on a pcap file containing TCP traffic. This was done in preparation to posting it on a mailing list. I thought I was pretty smart by writing a Scapy program to change the IP addresses and write the changes to another pcap file. But, before it was posted, someone examining it told me that the TCP checksums were incorrect. Why would the TCP checksums be incorrect if I'd only changed the IP header? Because the pseudo-header used in TCP checksums uses the IP address. Sometimes the most embarrassing lessons are the ones that stay with you the longest.



Let's look at TCP checksum processing. First, when dealing with IPv4, the IP checksum must be validated. This is to make sure that the packet did not get corrupted on its journey from the previous router to the destination host. The packet is silently discarded if the receiving host's checksum value does not match the checksum in the IP header.

The TCP layer performs this same validation process on the checksum value found in the TCP header. A matching value causes the data portion of the packet to be passed up to the application layer. A value that does not match is silently discarded. There is no reset returned by the receiving host. If the datagram is discarded, it is up to the sending host application or protocol layer to detect and deal with this.

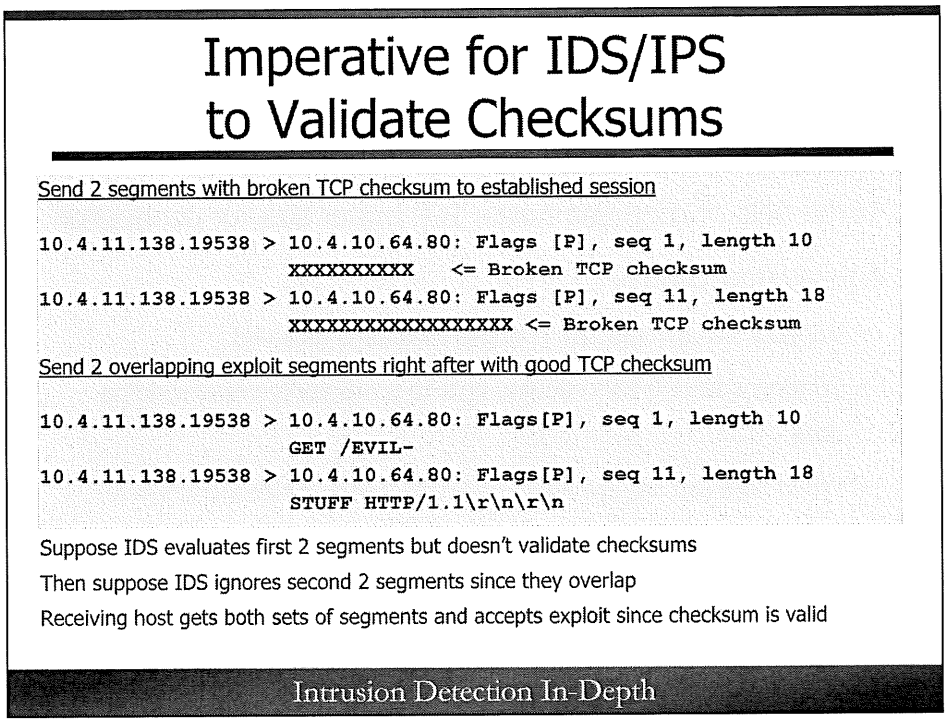


Let's examine a very specific example of how the pseudo-header protects against delivering the packet to the wrong host. Assume that we have a host that sends a TCP packet to destination IP 10.2.3.4. For the purposes of this example, it really doesn't matter if the transport layer is TCP or UDP since both use the pseudo-header. The transport layer checksum will include the pseudo-header fields in the checksum computation. Therefore, a destination IP of 10.2.3.4 will be used in the TCP checksum computation.

On its way from the sending host, the packet travels through a router that, as you remember, must validate the IPv4 checksum before forwarding it. Suppose the router validates the IPv4 checksum, decrements the TTL and now needs to compute the new IP checksum. Now, for some unforeseen reason, the IPv4 layer of the router somehow corrupts the destination IP to be 10.2.3.5. The IPv4 IP checksum is recomputed using the corrupted destination IP. The IPv4 IP header checksum will be valid so the packet will continue on towards the wrong destination IP, 10.2.3.5.

Now, assume that the IP 10.2.3.5 exists and it listens on the destination port found in the TCP header. The corrupted packet will arrive at the wrong destination IP. The IPv4 layer will validate the IP checksum and it will be correct since destination IP 10.2.3.5 was used in the IPv4 IP checksum computation by the corrupting router. The packet will be pushed up to the transport layer where it will use the pseudo-header fields in the checksum computation. But, the TCP checksum performed will use destination IP 10.2.3.5 in the corrupted packet for the pseudo-header computation and comparison against the packet's actual TCP checksum. However, this will not match the TCP pseudo-header checksum from the sending host that used 10.2.3.4 as the destination IP in the pseudo-header checksum. Host 10.2.3.5 will then discard the packet because the embedded protocol checksum does not match the computed checksum derived by the destination host.

While using the pseudo-header as a means to detect rare instances of corruption may be an esoteric concept, one thing to bring away from this discussion is the fact that some of the fields in the IP header are used in the computation of the TCP and UDP checksums. This becomes important when discussing Network Address Translation (NAT) since NAT will often change the original packet's IP address in the NAT'd packet. This means that not only will the IPv4 checksum have to be recomputed using the NAT'd IP address, but an embedded TCP or UDP checksum will also have to be recomputed since the pseudo-header will need to include the NAT'd IP address.

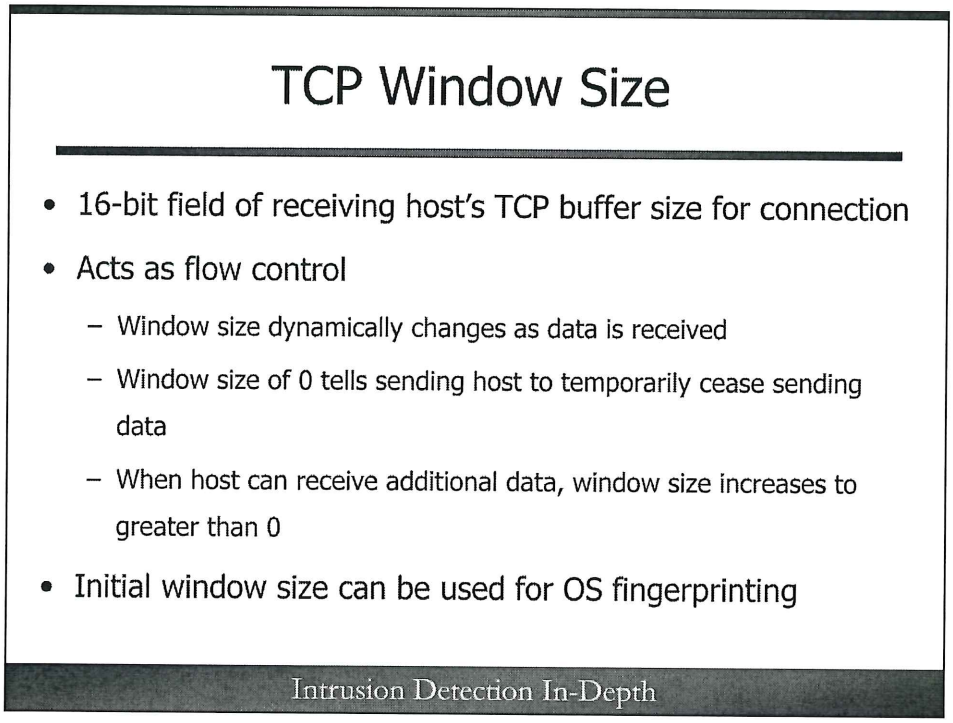


Perhaps you are wondering about the significance for an IDS/IPS to validate checksums. Let's take a specific example where we corrupt the TCP checksum so it is not valid. This particular example may be more pertinent for an IDS since an IPS may be more likely to just drop an overlapping packet that is not a retransmission.

Say there is an already established TCP session to a web server in the network that the IDS/IPS protects. For the purpose of this example, assume that "GET /EVIL-STUFF" represents malicious activity. We send "EVIL-STUFF" in two separate segments, splitting the payload in the middle of what is most likely the IDS/IPS signature of "EVIL-STUFF" content of some sort. The reason we do that is that the IDS/IPS must reassemble the two segments in order to trigger an alert. It is possible for an IDS/IPS to treat exploit content found in a single segment differently than those found in multiple split segments since it has to reassemble those individual segments. For instance, an IDS/IPS may alert on a single payload of "EVIL-STUFF" even if the checksum is not correct or the TCP sequence numbers are wrong for the session just because it sees exploit content. It's like catching the low hanging fruit. In other words – it is better to generate a false positive than a false negative.

First, we send the two segments that have the same payload length and TCP sequence numbers of the ones we use for the exploit segments. But these two segments act as kind of a decoy because they have invalid TCP checksums and innocuous payload. If an IDS does not validate TCP checksums, it does not alert or block these segments. But, the destination host that receives these segments will discard them because it validates the TCP checksums.

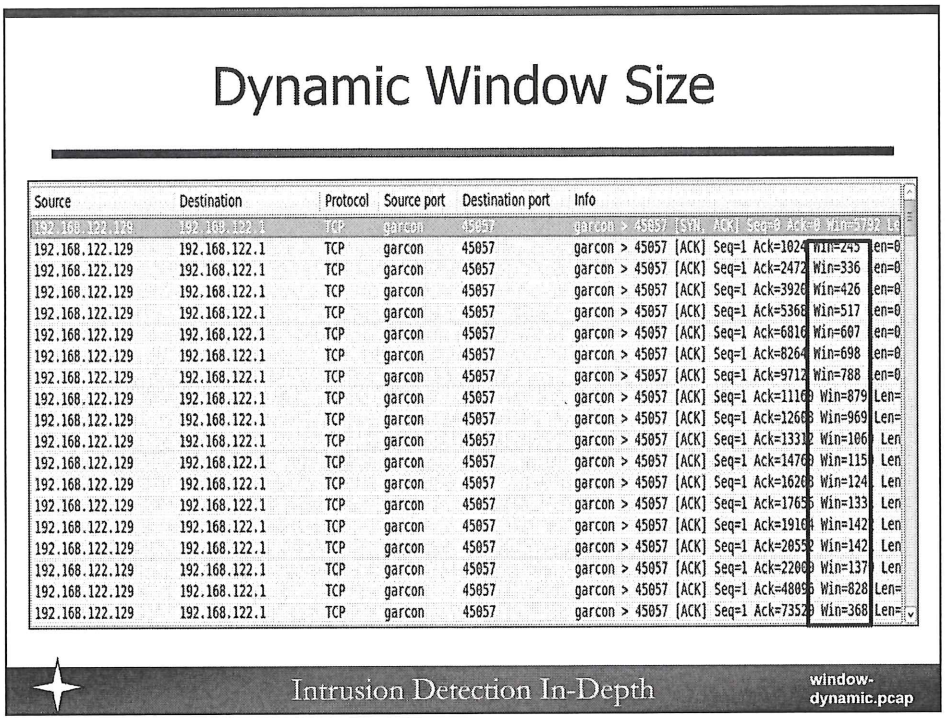
Now, let's finish the attack by sending the overlapping segments with exploit content and valid TCP checksums. If the IDS believes that it has already evaluated segments with identical TCP sequence numbers and content, it may ignore these segments. It is extra work for an IDS to re-evaluate traffic it has already seen, so some ignore overlapping segments. It is possible for it to permit these exploit segments to reach the destination web server undetected. This is just one possible scenario of attempting to evade an IDS that fails to validate checksums.



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 window size becomes smaller for all data that has been received but not yet processed by the receiving host. If the receiving buffer ever becomes full, the window size becomes 0, informing the sending host not to send any more data. Once the receiving host has processed some of the data and freed up room in the buffer for more, it will send 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 by use of the window size. We have a tendency to assume that the sender is really the one controlling the flow of data across the wire. But, for the most part, the receiver uses the window size as the director of the data flow.

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.



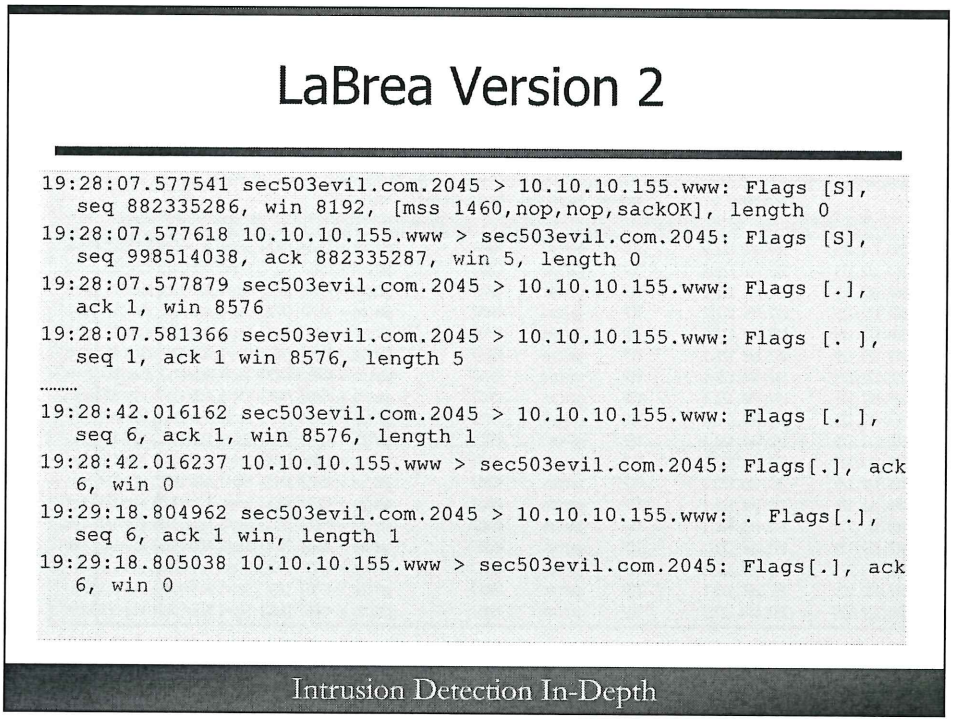
Let's examine the middle of a session where a Linux host 192.168.122.129 requests a large amount of data from 192.168.122.1. We examine the dynamic window size of 192.168.122.129 as it acknowledges data from 192.168.122.1 and adjusts its window size according to the amount of data it is able to process. The data packets from 192.168.122.1 causing the window size adjustment 192.168.122.129 are not included.

What you do not see, and what we have not yet covered is a TCP option window scale that makes the actual window size larger than the value seen in the window field of the TCP header. 192.168.122.129 announces its initial window size as 5792, yet we cannot see the window scale value of 5 that makes it 32 times larger than 5792. The window sizes appear small to begin with values of 245, 336, etc., but remember these values are multiplied by 32.

Initially, the window size appears to expand to a maximum of 1422 * 32 and then contracts as it has received more data that it needs to process. By decreasing the window size, it informs the sender to slow the rate or amount of data.

To display the output, enter in the command line:

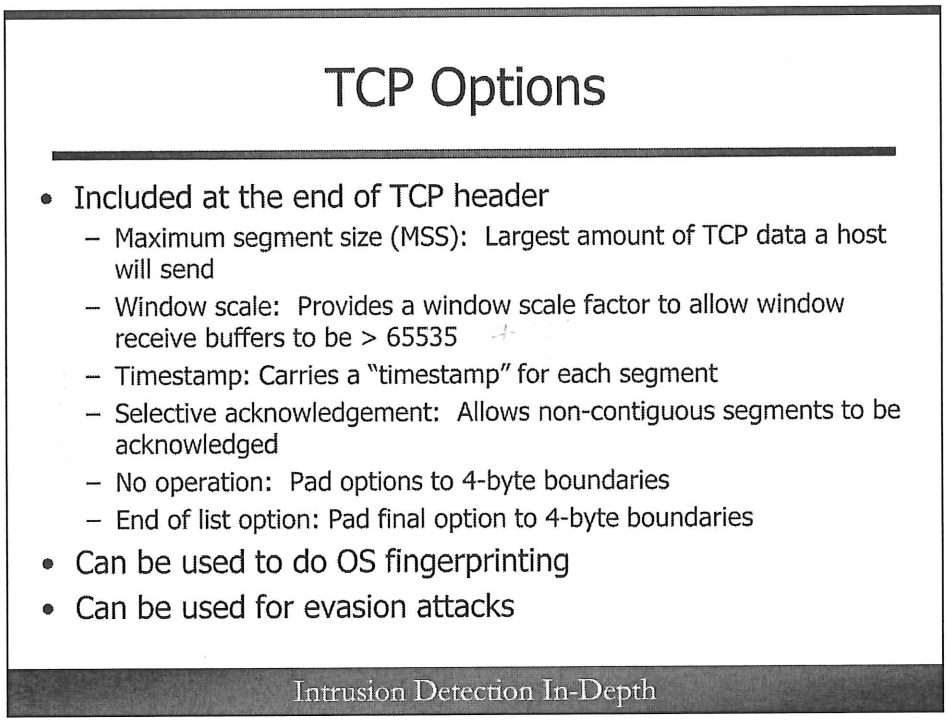
wireshark window-dynamic.pcap



If you recall, the original version of LaBrea was able to slow down a scanning or attacking host for the amount of time it took the attacker's TCP connection to timeout from a lack of a response after the three-way handshake. Depending on the attacker's TCP/IP stack implementation of the number of retries and the back-off time between timeouts, the attacker could be delayed several minutes.

LaBrea's author, Tom Liston, improved on his own concept using another technique known as the TCP persist timer. As we just learned, if a receiving host's TCP buffer (or window) is filled and it cannot accept any more data from the sender, it notifies the sender to stop by setting the window size to 0. Ordinarily, when the receiver's buffer frees up space after it is processed, a window update is returned to the original sender with a window size greater than 0. What if this new window update is lost? Both sender and receiver would be frozen waiting for the other to act. There is a mechanism to deal with this known as a window probe. After a timer expires and the sender has not received any new window updates from the receiver, the sender issues a window probe TCP packet that carries 1 byte of payload with the exclusive purpose of soliciting a response from the receiver to discover if the window size has been increased. The sender will persist in sending window probes until the window size increases or until the session terminates due to inactivity, hence the origin of the name persist timer.

The new version of LaBrea uses the persist timer to tarpit the attacker for an indefinite amount of time. It works exactly like the previous version of LaBrea up through the three-way handshake. Instead of not responding, LaBrea reacts to the data sent with an acknowledgement, but with a window size of 0. It doesn't increase the window size via an update, forcing the attacker's host to send a window probe. The LaBrea host responds to the window probe, but again advertises the window size as 0. This pattern of a window probe and a response of a window size of 0 continues indefinitely. This tarpits the attacker's host into an extended connection with the LaBrea host if there is no intervention.



TCP options are optional parameters that are stored at the end of the TCP header. The MSS should be sent on the SYN only and it represents the largest TCP payload that a host will send. Remember that the total IPv4 datagram will be at least 40 bytes more than this accounting for the IP header and the TCP header. This is set once and not readjusted like the window size. The optimal value for this is very close to the MTU minus 40. This doesn't account for header options, though. If an MSS value is not announced, the receiving host will assume that it is 536 bytes. The default MTU for IPv6 is 1280.

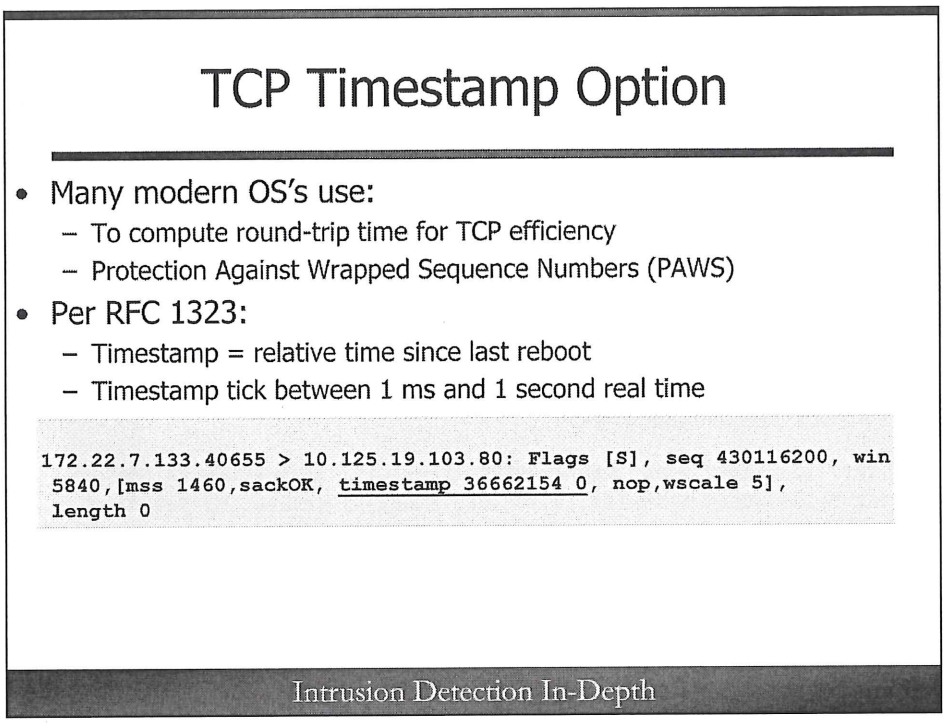
The window scale is a way to allow a receiving TCP buffer to be larger than the 16 bit maximum value of 65535. The window scale value indicates that the TCP window size be increased by 2^x. For instance, Vista may set the TCP window size to 8192 with a window scale of 8. Essentially the TCP window size becomes $8192 * 2^8$ or $8192 * 256$ or 2,097,152. Another way to think about this is that you take the value of 8192 and shift it left 8 bits.

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 can also be used to make sure a reused and old sequence number does not accidentally get included with a current exchange.

A newer option is the selective acknowledgement. Both sending and receiving hosts have to support this option for it to be used. If the selective acknowledgement is not used a host can acknowledge only the last chronological sequence number it received. In other words, if a TCP segment is either lost or late in arriving, subsequent arriving segments may not be acknowledged. The selective acknowledgement, like the normal acknowledgement, informs the sending host of the next expected sequence number. However, using the selective acknowledgement, it can also explicitly inform the sending host of the beginning and ending sequence numbers of any out-of-order packet(s) that it has received. This informs the sending host to re-send the missing only that and not others that have been acknowledged via the selective acknowledgement.

A couple of TCP non-functional options are supplied to pad the options to the required 4-byte boundaries. The 1-byte NOP can be used to pad in the middle of the options list, and the 1-byte EOL option is used to pad and indicate the end of options.

The TCP timestamp in a segment must be greater than or equal to the previous in-order one sent. If an IDS/IPS does not take this into consideration, it is possible to cause an evasion in the same manner as was demonstrated for failure to validate checksums. The receiving host will reject a segment with an invalid checksum and the IDS/IPS should too.



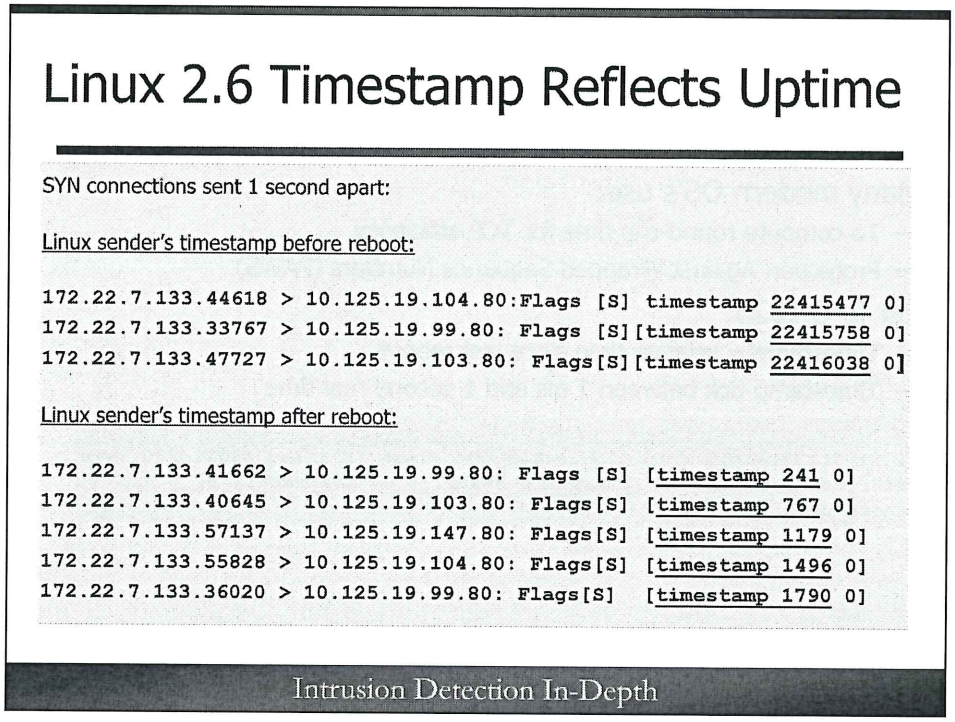
The TCP timestamp option is defined in RFC 1323 "TCP Extensions for High Performance". Its purpose is to maintain a value representing a notion of the host's current time in a field in the TCP options. Actually, the timestamp field contains both the sender and receiver timestamps. This permits either 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.

Additionally, the timestamp is used for a mechanism known as Protection Against Wrapped Sequence Numbers (PAWS) that 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. Frankly, this probably seemed like a good idea when the Internet was a nicer place; however, it is such a complex mechanism it introduces ambiguities that can be used for evasion purposes.

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 since this is a SYN packet and no communication has occurred with the receiver yet.

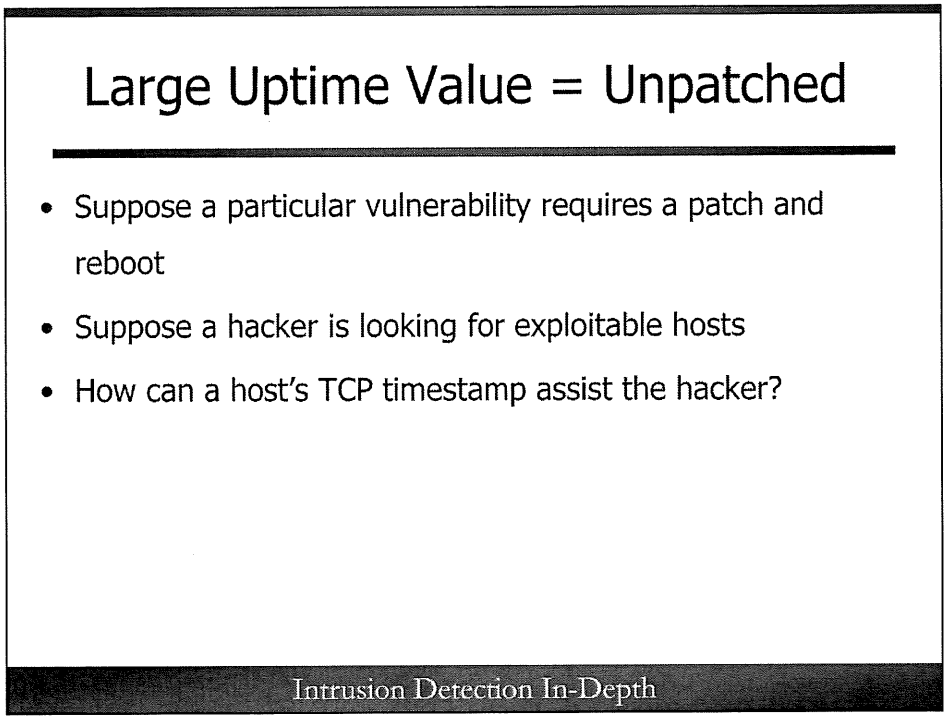
If you care to learn more about TCP timestamps, RFC 1323 can be found at:

http://www.ietf.org/rfc/rfc1323.txt



As an example of what the TCP timestamp looks like on a Linux 2.6 system before and after reboot, take a look at the above values. The SYN segments above are sent a second apart from client 172.22.7.133 to different servers on network 10.125.19/24. The tcpdump timestamps (reflecting the capture time of the packet) and other tcpdump information have been removed to fit the pertinent data on the slide. The sender's timestamps range from 22415477 through 22416038 in the three segments before reboot.

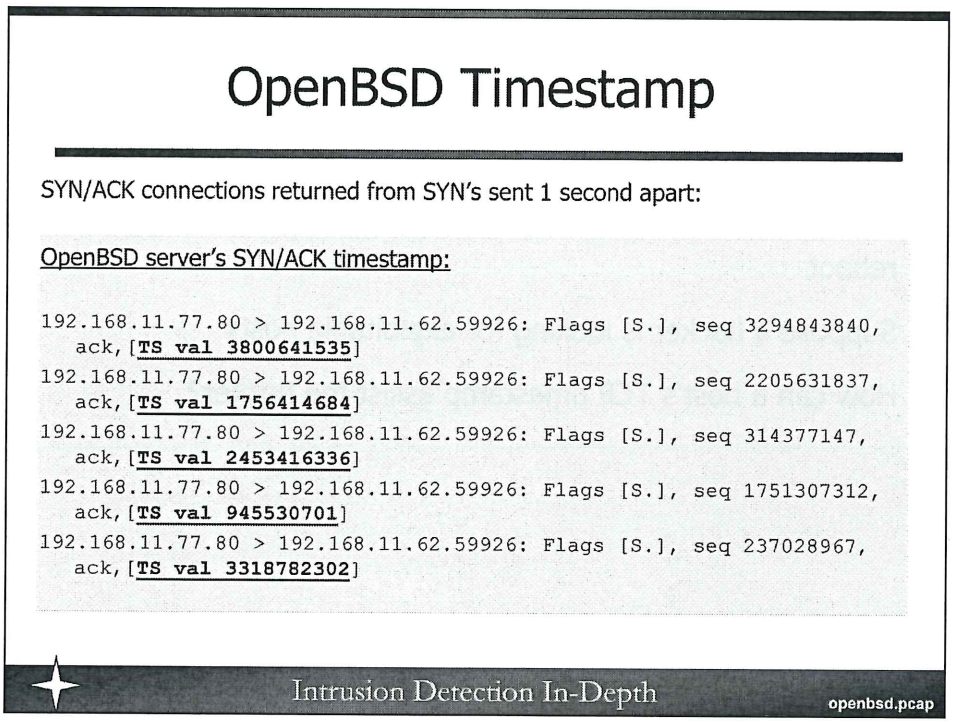
Now take a look at the recycled values after reboot that range from 241 through 1790. Again, the host 172.22.7.133 sends 10.125.19/24 hosts a SYN segment a second apart. While the host used above was a Linux 2.6 operating system, this same behavior is present in many other operating systems. Let's see how this may help an attacker.



Why should you care about TCP timestamps? As mentioned in a previous slide, they provide a mechanism for IDS/IPS evasion. They also signal to a would-be hacker whether or not you've rebooted your system lately.

Imagine the following scenario. A Microsoft update is released on Microsoft Tuesday for a critical bug that requires a reboot of the host. Now, suppose that there is an exploit available for the particular vulnerability. If a hacker is attempting to be selective about her/his target, the TCP timestamp would offer a pretty good clue soon after the patch was released whether or not a potential target host has been patched. In this instance, the TCP timestamp offers too much information we'd rather not disclose.

A hacker interested in exploiting a particular target may watch traffic to or from the target around the time of the update and observe whether or not the timestamp value is less than a previous time. A timestamp that has increased since last observed probably has not been patched and rebooted. A lower timestamp means a reboot was likely. It is not likely that a timestamp would wrap its value back to 0 without a reboot since it is a 32-bit value that could possibly represent years in real time value before wrapping.



OpenBSD developers realized that randomization of values is a good thing. In fact, they not only randomize TCP timestamps, but TCP Initial Sequence Numbers (ISN), ephemeral ports, and IP identification numbers. The less predictability in these values, the harder it is for an attacker to gain any valuable knowledge about the host to facilitate an attack.

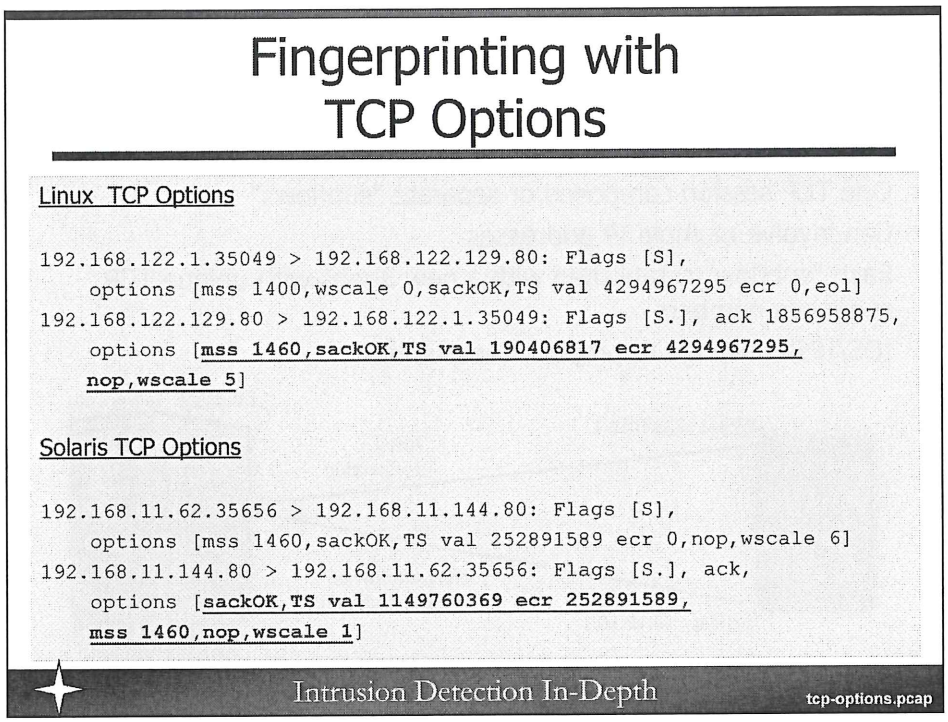
Let's look at what OpenBSD has done. The above traffic is from an OpenBSD web server, 192.168.11.77, responding with a SYN/ACK to SYNs sent approximately a second apart from 192.168.11.62. The timestamps returned are not at all predictable; they have been randomized. You may wonder if this breaks the intentions of RFC 1323. The timestamps values are randomized for each new TCP session, however the timestamps within a given session are incremental and relative to the beginning of the session. This allows the host to perform accurate round trip time measurements and perform calculations for Protection Against Wrapped Sequence Numbers as well. This seems to be the ideal solution for implementing the original provisions for TCP timestamps without divulging any details about the system's uptime.

To read more about OpenBSD's stack:

http://www.securityfocus.com/columnists/361/2

To display the output, enter in the command line:

tcpdump -r openbsd.pcap -nt

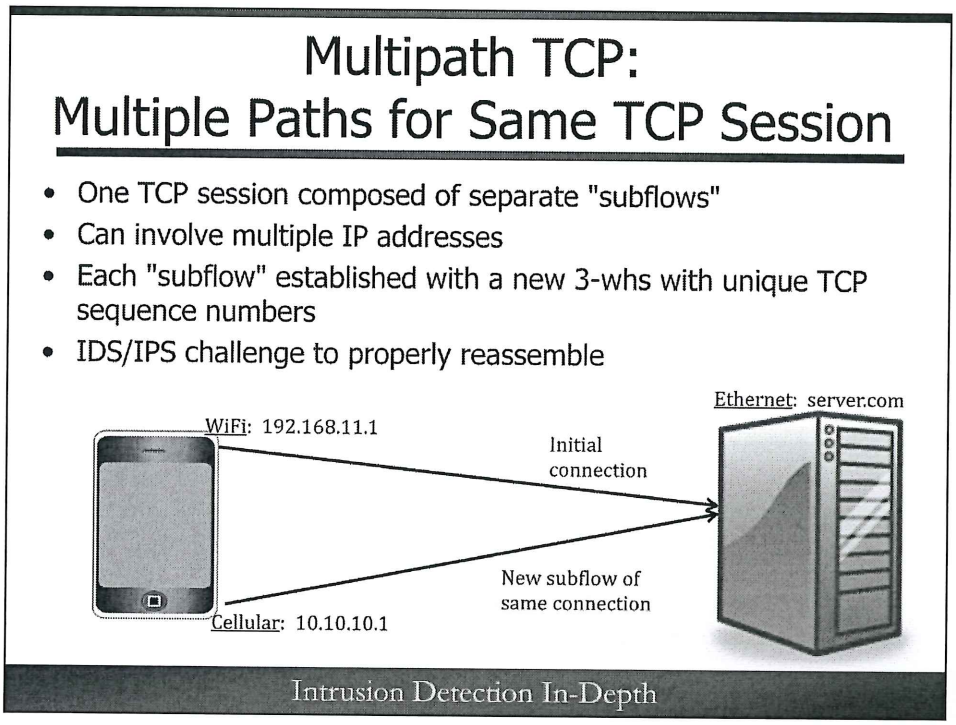


One of the ways that Nmap fingerprints an operating system is by sending different TCP options to the host to see how it responds. Not all operating systems support all the TCP options so this assists in operating system identification. Also, different operating systems will store the TCP options in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

The first connection above is to a Linux host 192.168.122.129. It supports a MSS of 1460, the selective acknowledgement, timestamp, and window scale TCP options. The second connection is to a Solaris host 192.168.11.144. It too supports the same TCP options. But what is different about it? The order that the TCP options are stored in the header differs. Solaris is one of the few operating systems that does not store the MSS as the first TCP option. The window scale values are different as well.

To display the output, enter in the command line:

tcpdump -r tcp-options.pcap -nt

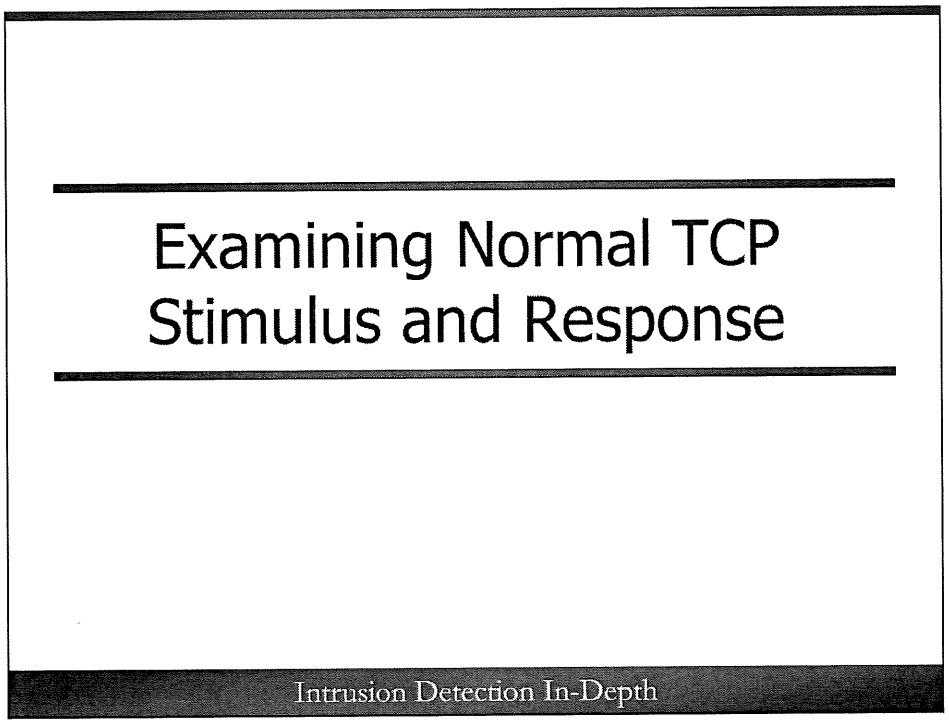


As if TCP were not complex enough, a new "brand" of TCP known as multipath TCP has been introduced. When TCP was conceived many years ago, there was a single network interface per host/device to communicate with other hosts/devices. It is common now to find that devices have multiple network interfaces for communication. Take for instance the cell phone; it may have interfaces to communicate using WiFi or cellular networks. What if it were possible to use both those interfaces in the same TCP "session" for efficiency and redundancy in case one communication path becomes unavailable during the session? That is what multipath TCP is all about.

The way it works is that the client sends a SYN with a unique Initial Sequence Number (ISN) containing a new TCP option number 30 that has a subtype of MP_CAPABLE (value of 0), indicating that the device is multipath capable, to the server. The server responds with a SYN/ACK and a unique ISN and uses the same TCP option and subtype values if it too is multipath capable. The final ACK of the 3-whs has the same TCP option and subtype values. Now, the client can use other interfaces for communication in the same session by creating a new "subflow" that belongs to the initial TCP session. It does so by sending a SYN with a unique ISN, including the TCP option number 30, but now with a subtype of MP_JOIN (value of 1), indicating the desire to join the same session. Both the initial connection and any subsequent joining sessions use an exchanged set of cryptographic keys for security and association for the same TCP session. Another subtype of the TCP option 30 is known as the data sequence signal (DSS) - subtype 2 that is used to indicate how a particular subflow's data fits in the entire session. Once the subflows are established, the client and server decide which subflow is to be used for a given exchange.

The reassembly of these multiple subflows into a single TCP session is a challenge to any security device such as an IDS/IPS. As with any protocol, the IDS/IPS would have to understand this very different and complex protocol in order to do any reassembly. Any sensor would have to capture traffic from all the interfaces involved to perform the reassembly. Until security devices become more sophisticated, this is a real evasion challenge. See the following discussion on multipath TCP security issues:

http://threatpost.com/multipath-tcp-introduces-security-blind-spot/107534



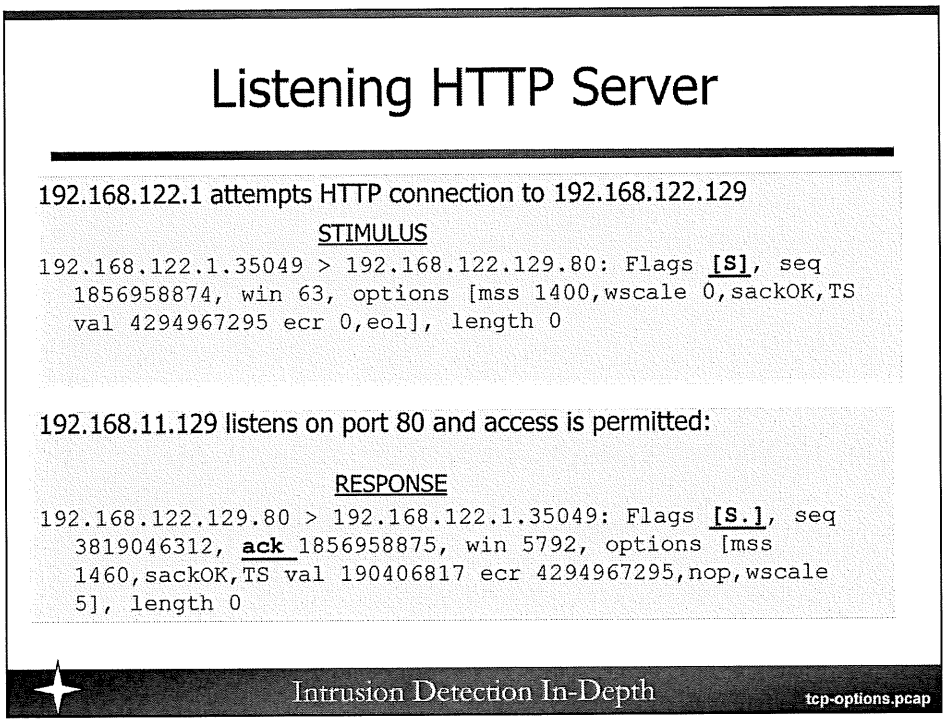
Now that you are familiar with TCP theory, it is time to explore the nature of TCP in the context of what is a normal, or perhaps better described as an expected, response when a given stimulus is sent. As mentioned several times before – you have to be aware of what is considered normal in order to determine what is unusual or abnormal.

TCP Stimulus - Response

- Using TCP as an example protocol, let's examine various situations for expected responses
- We 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

Intrusion Detection In-Depth

Let's look at responses to an attempted port 80 connection under varying conditions. In the following slides, we will show some of the varied and expected responses to the identical stimulus. Obviously, this is not an exhaustive list of all conditions that may be encountered with an attempted port 80 connection. These particular conditions have been selected for illustration because they are some of the most common.

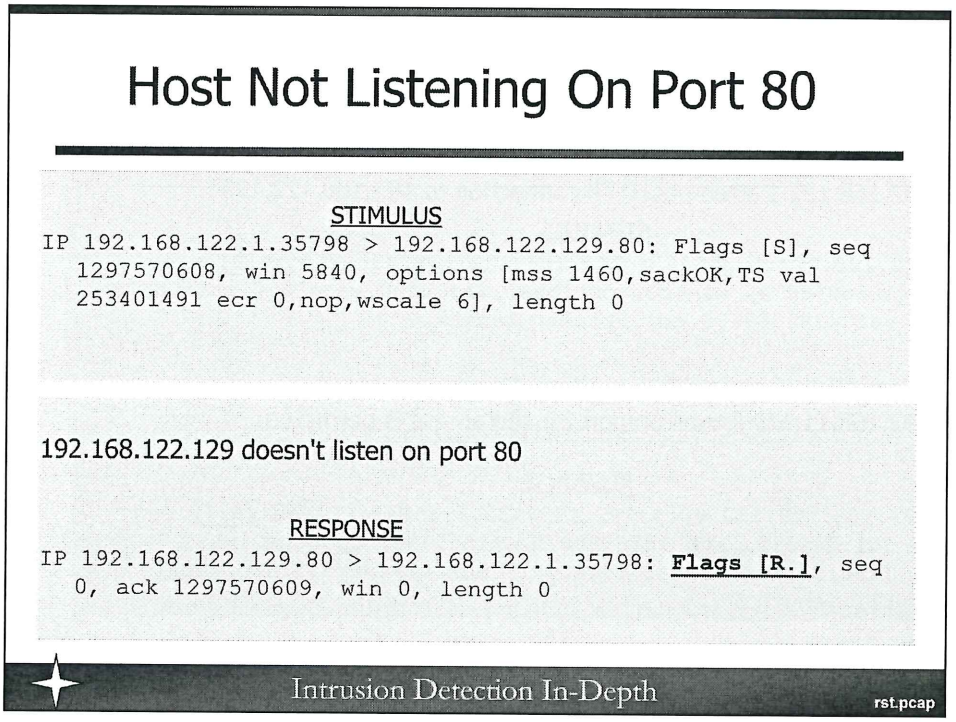


What is the expected response when client host 192.168.122.1 attempts to connect to port 80 on listening server 192.168.122.129? We've already discussed the concept of the three-way handshake for TCP session establishment. If you remember, the first part of the process requires that the client initiates a TCP connection with the SYN flag set to the server—this signals the desire to connect. In this slide, we 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, we see the expected response from 192.168.122.129 with a SYN/ACK flag combination. This says that 192.168.122.129 is listening at port 80 and is capable of establishing this HTTP connection.

To display the output, enter in the command line:

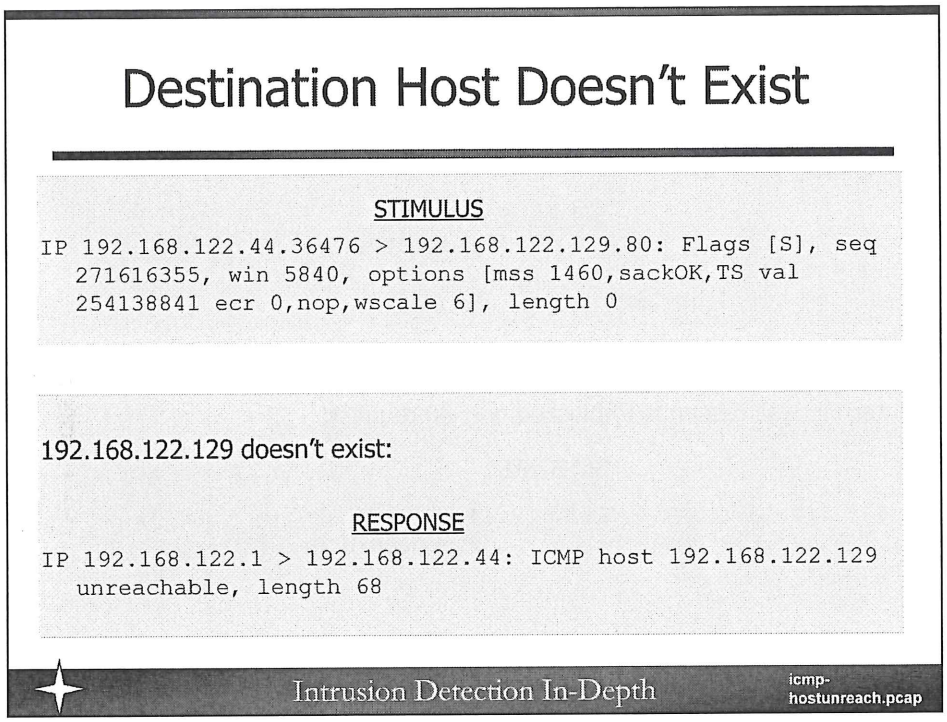
tcpdump -r tcp-options.pcap -nt



We 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 a RESET/ACK that is an abrupt termination to the connection.

To display the output, enter in the command line:

tcpdump –r rst.pcap -nt

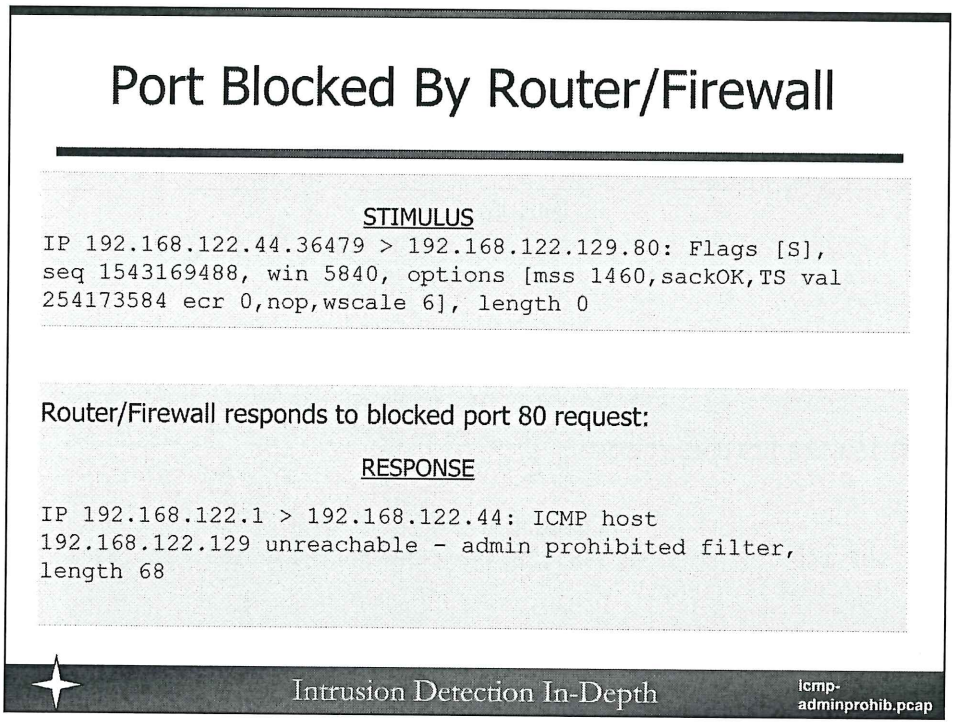


What happens if 192.168.122.44 attempts a port 80 connection to 192.168.122.129, but 192.168.122.129 doesn't exist? If this is on the same broadcast domain, ARP will be unable to resolve the IP address and the failure causes the client to conclude that there is "No route to host".

However when a router or firewall exists where a host is incapable of responding, 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 in the command line:

tcpdump –r icmp-hostunreach.pcap -nt

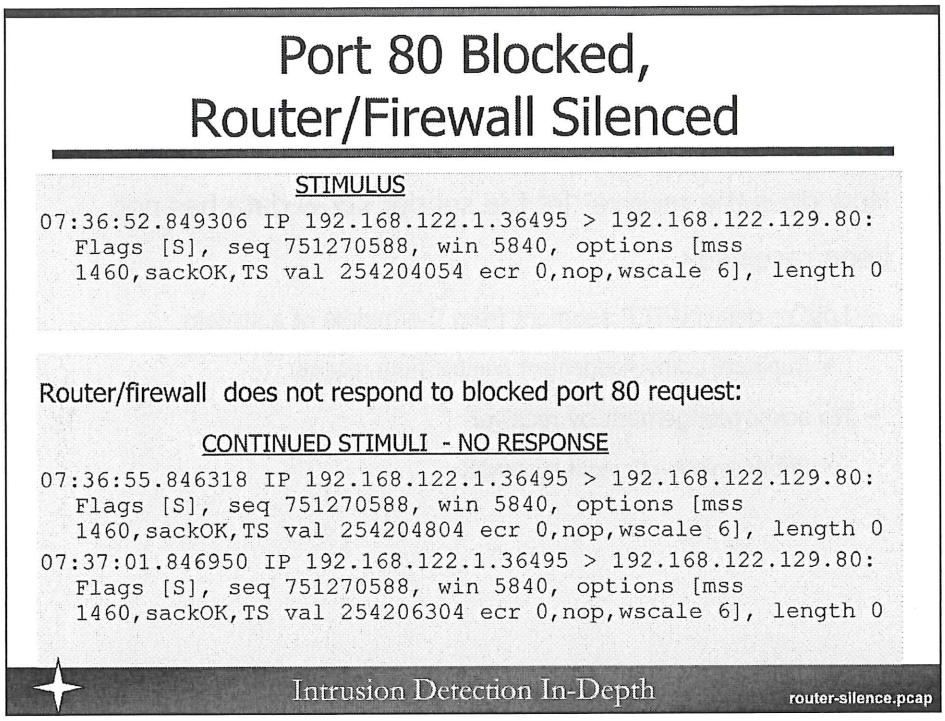


What if the port 80 is blocked by a firewall or router? What kind of response will you see? Again, the router for 192.168.122.1 may attempt to inform 192.168.122.44 that 192.168.122.129 is unreachable and further add that this is because of "admin prohibited filter," meaning that the access was blocked.

While 192.168.122.129, was just trying to be helpful and informative in the last two situations examined, it is giving out some valuable information if someone is probing your network. It's possible to silence some firewalls/routers from issuing unreachable messages to silence the router and limit the information that it divulges.

To display the output, enter in the command line:

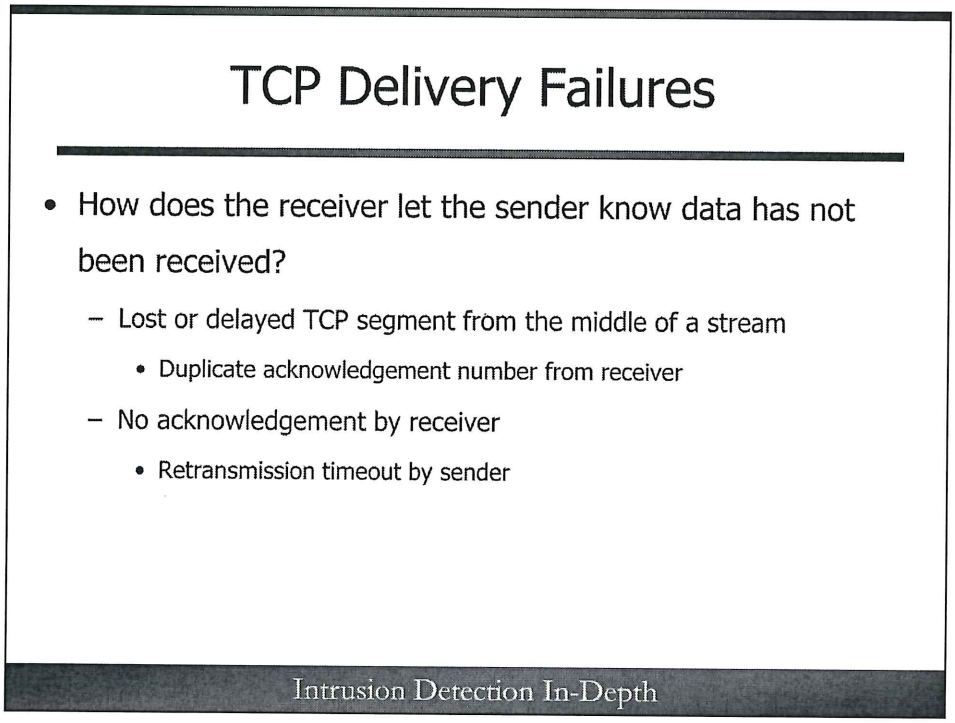
tcpdump -r icmp-adminprohib.pcap -nt



Now, we 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 will continue 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 will give up on the connection after it has exhausted the maximum number of retries.

- To display the output, enter in the command line:

  tcpdump -r router-silence.pcap -nt



We will now look at TCP delivery failures in more detail. We've seen that it is up to the receiver of data to inform the sender that the expected data has not been received. If a TCP segment is lost in the middle of a stream, the receiver should be able to detect this. The receiver keeps track of the next expected sequence number.

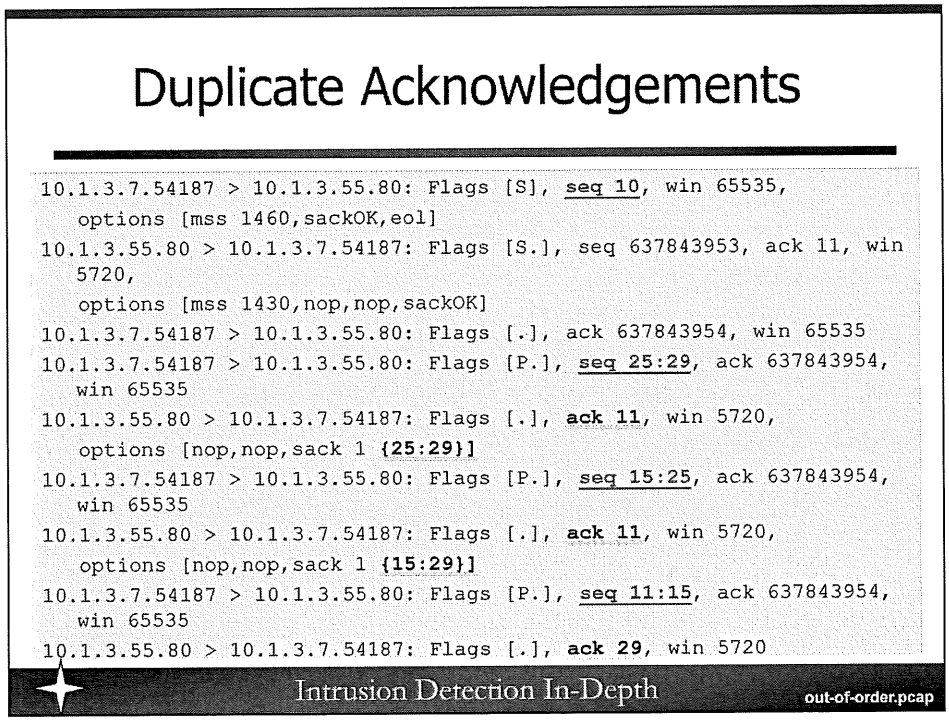
If the receiver gets a TCP segment with a sequence number greater than the one it expects, it will not acknowledge the out-of-order sequence number. Instead, it will send back an acknowledgement of the next expected sequence number. Even when the TCP option known as the selective acknowledgment is used, 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.

It can take up to three duplicate acknowledgement numbers of the expected sequence number to convince the sending host that there is a lost packet.

Alternatively, 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 some kind of acknowledgement before a timer expires, it will retransmit the unacknowledged segment.

What is the value of 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 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.

no ack
last segment is lost



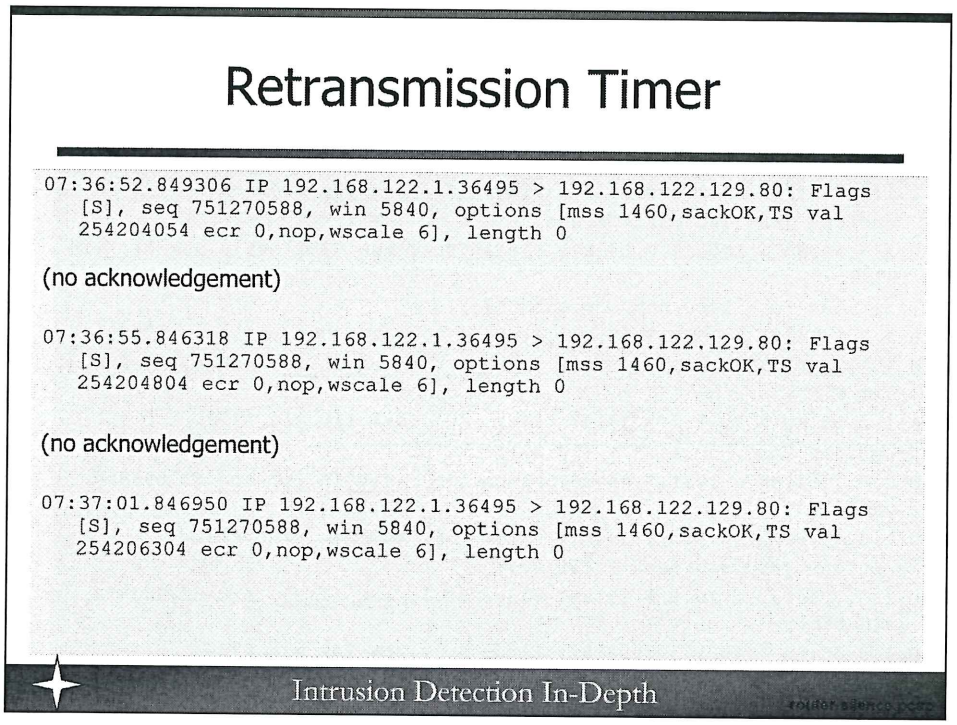
Let's see how duplicate acknowledgements work. We are looking at an exchange between hosts 10.1.3.7 and 10.1.3.55. Both hosts support the TCP selective acknowledgement option. We'll see how this assists in the reporting of duplicate acknowledgements.

The tcpdump output reflects the use of absolute sequence numbers for clarity. The client's sequence number values have single underlines while the server's acknowledgements of those sequence number values are bolded with a double underline. The three-way handshake begins with a SYN from 10.1.3.7 with a crafted TCP sequence number of 10. The server acknowledges this with an ACK value of 11. The client sends out-of-order sequence numbers of 25 through 29 in record 4 above. The server was expecting sequence number 11. The server reports back in record 5 using the acknowledgement value that the next expected sequence number is 11, but it also reports by virtue of the selective acknowledgement TCP option that sequence number 25-29 were received.

The client sends more out-of-order sequence numbers 15-25 in record 6. Again the server responds with a duplicate acknowledgement of TCP sequence number 11, and updates the selective acknowledgement range to 15-29 as received sequence numbers. Finally, in line 8, the client fills in the gap of 11-15 to complete the range of 11-29. The server acknowledges this with an acknowledgement value of 29.

To display the output, enter in the command line:

tcpdump -r out-or-order.pcap -ntS



We've seen a couple of instances already where the retransmission timer is used when a client attempts to connect to the server, yet there is no response. The number of retries of the same connection and the time that elapses between each successive retry is dependent on the TCP/IP stack. Eventually, the sending host will give up trying to connect. How do we know that these are retries of the same connection and not completely new connections? Look at the source ports and the TCP sequence numbers. They remain the same for all of the connection attempts. More likely than not, this indicates retries.

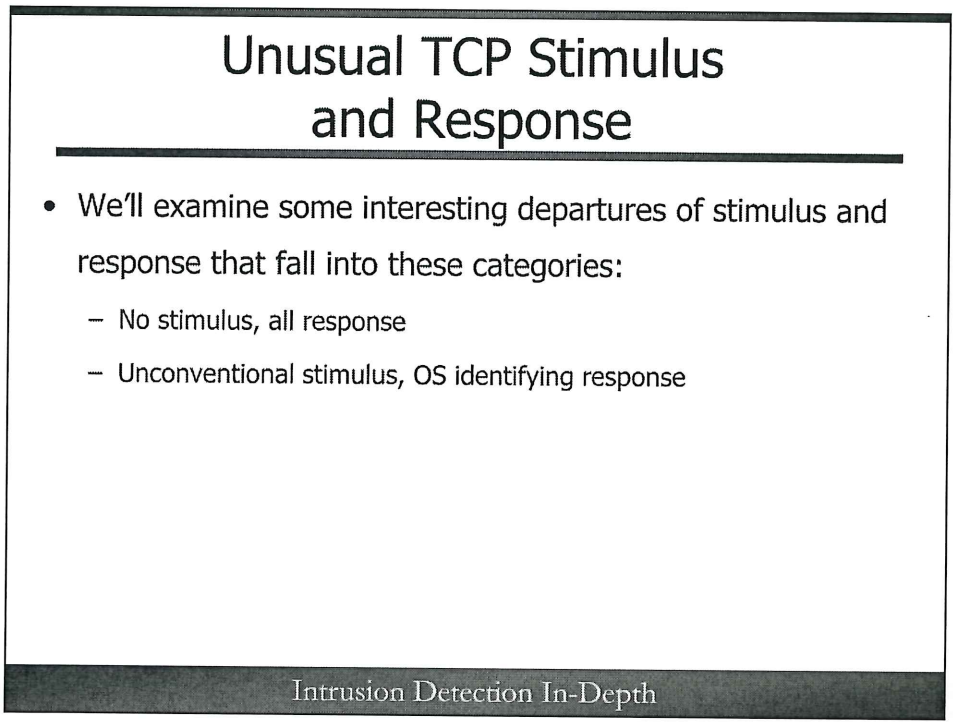
- To display the output, enter in the command line:
  tcpdump -r router-silence.pcap -nt

Normal TCP Stimulus and Response Review

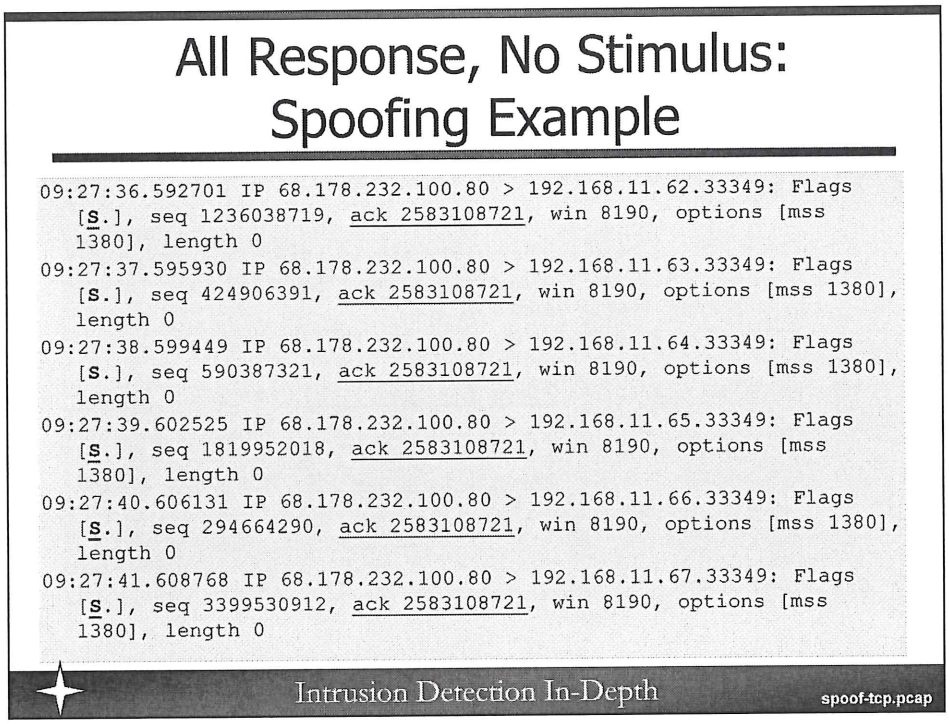
- Expected responses to attempted protocol connections are distinct depending on:
  - The availability of the requested service
  - A host's/router's ability to respond
- Some applications or programs deviate from the standard protocol behaviors

Intrusion Detection In-Depth

We have learned that there are unique responses for the same stimulus depending on the circumstances and availability of the requested service. Responses are also dependent on a host's or router's/firewall's capability to respond to a particular connection. Each of the different protocols has different expected responses.



We'll now examine some of the anomalous behaviors that hackers will throw your way in this section. These behaviors have many purposes and we'll examine what they might be with each of the categories we cover. These categories and anomalies are not all-inclusive; you may find many more.



This slide introduces the idea that sometimes you'll receive traffic in your network that appears to be a response of some sort. However, if you are able to check for any previous outbound activity that may have elicited the response, you will find none.

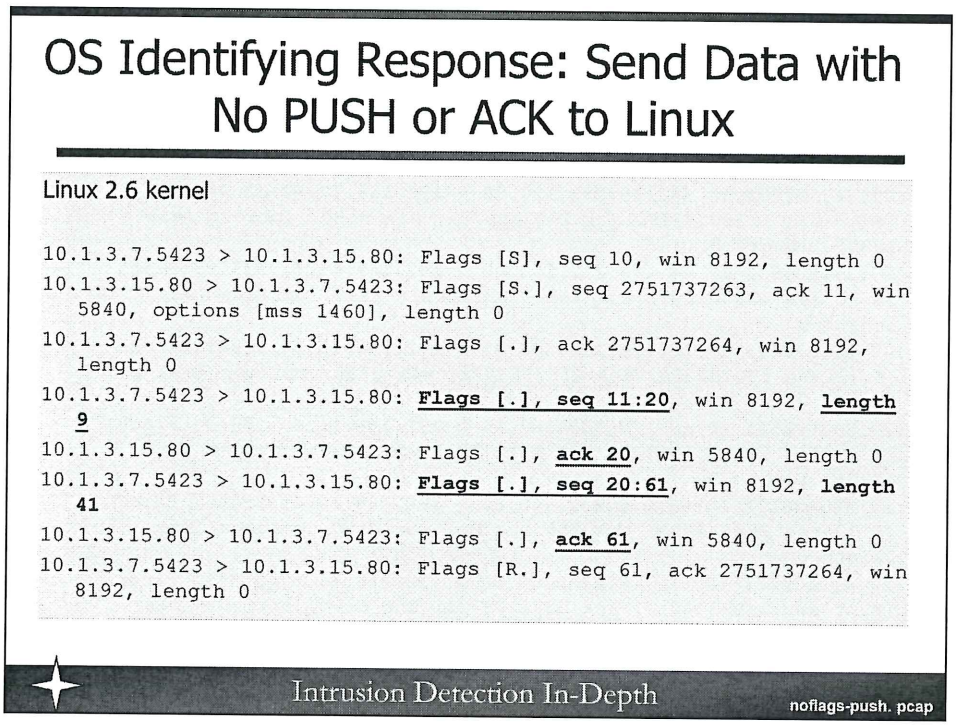
One reason that you may see responses with no initiated requests is that someone may be borrowing or spoofing your IP numbers for malicious use. This malicious use may be a denial of service against someone else using your IP's or perhaps using your IP's to hide as a smoke-screen and disguise the real source of malice. In fact, the Nmap scanning program has an option to use "decoy" source IP numbers while scanning with the intent to confuse or at least to cloud the real source IP. Many scanning programs these days have an option to select a spoofed source IP(s).

We see many records coming from sec503.com, 68.178.232.100, directed to many 192.168.11.0/24 hosts on our network. Note a source port of 80 and note the SYN and ACK flags set. It is unlikely that sec503.com is attempting a scan of 192.168.11.0/24 hosts using a SYN/ACK scan. An unsolicited SYN/ACK should elicit a RESET from a live host regardless if the port is active or not. This could be a mapping attempt; however, there may be a more plausible explanation.

A more likely scenario is that the192.168.11.0/24 hosts have been spoofed by some unknown source and port 80 requests are sent to sec503.com. If sec503.com offers 80, it will then respond with a SYN/ACK from source port 80 to the spoofed 192.168.11.0/24 hosts. You could check for outbound SYN segments on your network from the 192.168.11.0/24 hosts that would have elicited this response to validate whether or not the SYN segments were spoofed.

You may hear the term "backscatter" to refer to the unsolicited SYN/ACK segments.

To display the output, enter in the command line:
tcpdump -r spoof-tcp.pcap -nt



The Linux TCP/IP stack has some unconventional behavior that can be used to identify it. For instance, after the three-way handshake, it will accept a segment that contains payload, yet has no TCP/IP flags set.

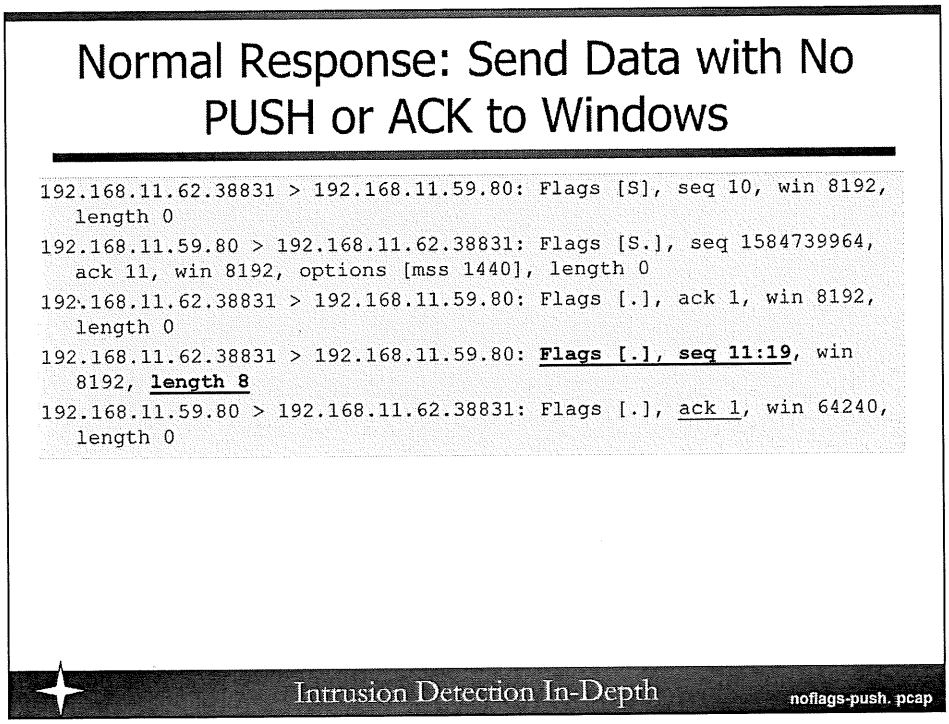
Above you see a three-way handshake to enable 10.1.3.7 to communicate with 10.1.3.15, a Linux host with a 2.6 kernel, on port 80. On the 4th line you see that the client sends 9 bytes of data, yet there are no flags set. 10.1.3.15 accepts this payload data as manifested in line 5 where you see an ACK of 20 – the next expected sequence number from 10.1.3.7. This is contrary to RFC 793 guidance that advises that the ACK flag be set for all packets except for the initial SYN.

In the 6th line above, the client sends 41 bytes of data – again on a segment with no TCP flags set. And, once again the Linux host acknowledges receipt of the data.

This is an unusual stimulus and even more unusual response. No other well-known operating system acknowledges a data payload when no TCP flags are set.

To display the output, enter in the command line:

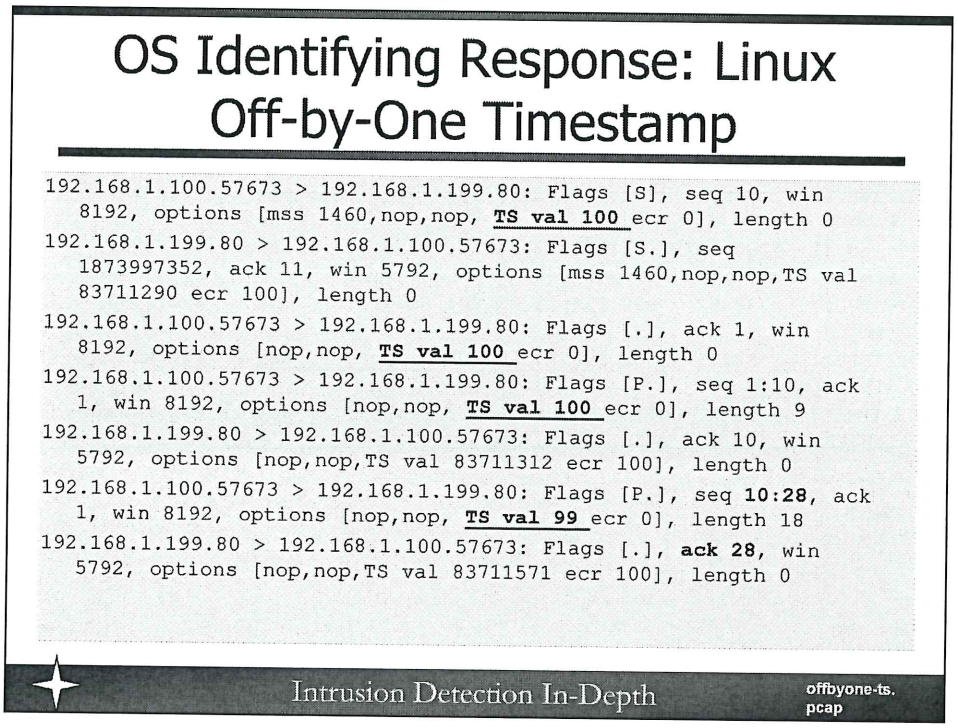
tcpdump -r noflags-push.pcap -ntS



In contrast, let's look at how the Windows host 192.168.11.59 responds. Once again the client attempts to send 8 bytes of data displayed in line 4 with no TCP flags set. The Windows server responds as expected, issuing a duplicate acknowledgement of the previously accepted TCP sequence number, conveying that it does not acknowledge the 8 bytes of data.

To display the output, enter in the command line:

tcpdump -r noflags-push.pcap -nt

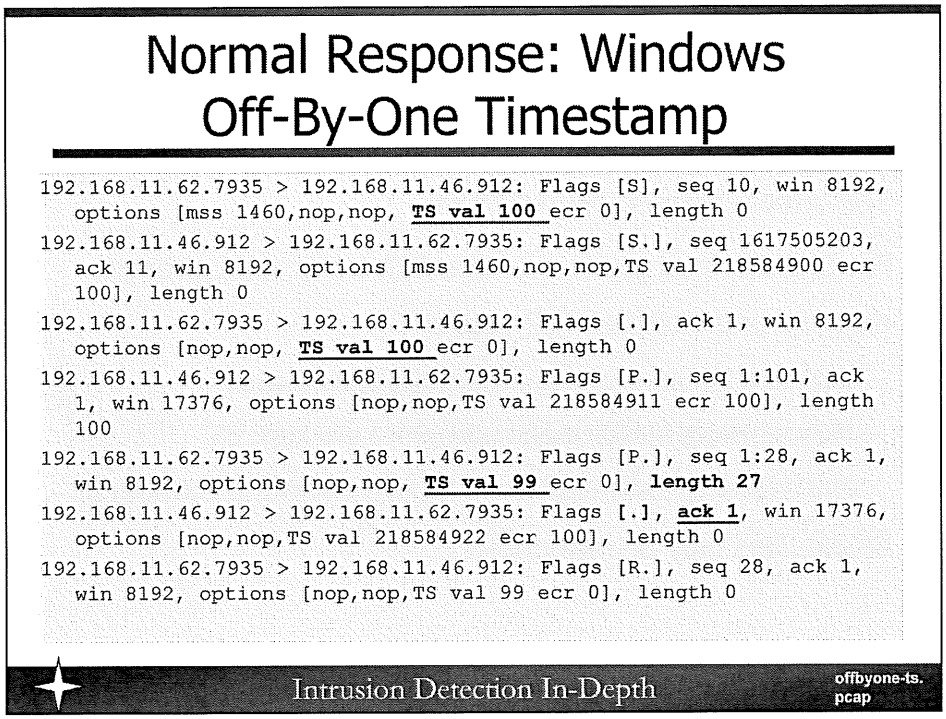


The way things are supposed to work is that an in-order segment is acknowledged only if its timestamp is greater than or equal to the last one sent by that side of the conversation. For instance, client 192.168.1.100 connects to 192.168.1.199 port 80. The timestamp value on the SYN is 100. This has been crafted along with the TCP sequence number of 10 to simplify and have less visual clutter using small values.

The client's timestamp remains at 100 for the ACK and the first PUSH of data in line 4. However, in line 6 the TCP timestamp is lowered to 99 and relative TCP sequence numbers of 10-28 are consumed sending 18 bytes of data. Yet, in line 7 the Linux receiver acknowledges all the data sent. This should not happen because the timestamp value of 99 is not greater than or equal to 100.

To display the output, enter in the command line:

tcpdump -r offbyone-ts.pcap -nt

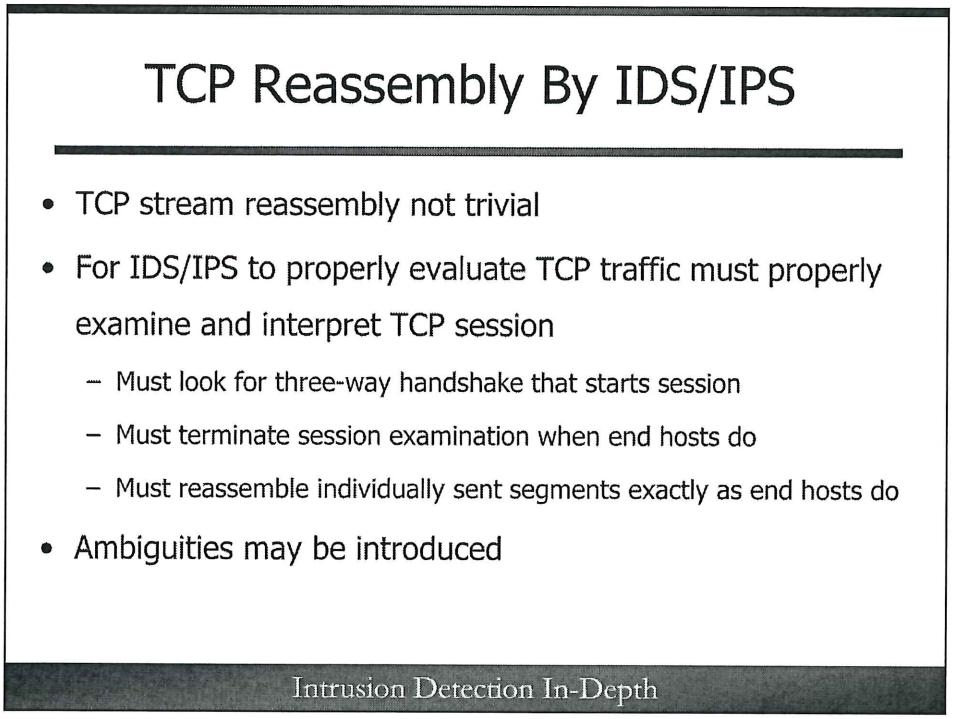


Here is a set of similar traffic, yet this time there is a Windows server. Once again, the client SYN and ACK has a timestamp of 100. And, again in line 5, the client sends 27 bytes of data with an invalid timestamp of 99. Windows sends a duplicate acknowledgement for relative sequence number 1, indicating that it did not accept the data.

We've seen two instances of unique behavior from Linux hosts that can be used to identify the remote operating system.

To display the output, enter in the command line:

tcpdump -r offbyone-ts.pcap -nt



So, now that we understand a little bit about TCP transmission, let's think for a moment what an IDS/IPS must do to examine and evaluate TCP payload in order to determine malicious content. The bottom line is that the IDS or IPS must take the TCP stream and reassemble it exactly like the destination host does.

Obviously an IDS/IPS must begin tracking a TCP session at its inception. But there are instances when there may be some doubt about the best way to proceed. What happens when an IDS/IPS does not see every segment involved in the three-way handshake because it was overloaded and may have dropped a segment? Should it begin tracking when it appears that the session has been established and hosts are sending and acknowledging receipt of data?

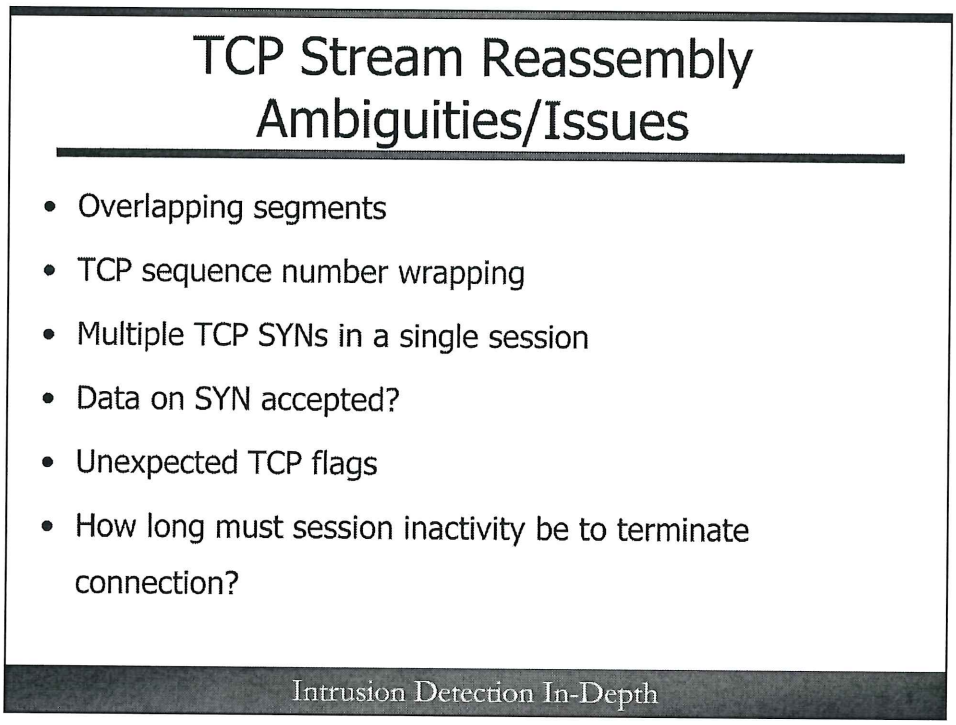
And when should an IDS/IPS stop tracking a particular TCP session? Sure, it's an easy choice when the session cleanly terminates either by an exchange and acknowledgement of FINs or a single RESET. But what about half-closed sessions when one side has signaled that it will no longer send any data by sending a FIN, but can still receive data?

Just as important – after the session has been established, the IDS/IPS must properly reassemble individual segments, including payload to interpret the entire stream content. It must examine and re-order all TCP sequence numbers including those that arrive out of chronological order. It must examine TCP flags for the purpose of the given segment. In addition, the IDS/IPS must validate the TCP checksum and discard segments that contain invalid checksums, as the receiving host will discard them. If there is a true IDS and not IPS, it must examine the TCP acknowledgement number to confirm the receipt of a sent segment. Finally, if TCP timestamps are used, the IDS/IPS must examine them to verify that the segment is current – otherwise, if it is old, it must be discarded as well.

Many of these different aspects of TCP segments and reassembly can introduce ambiguities where a given operating system favors a particular segment or discards a segment that perhaps another operating system

might keep. If the IDS/IPS does not interpret or resolve the ambiguity in the same manner that the destination host does, then malicious TCP traffic can evade detection by the IDS/IPS.

We will discuss evasions in greater detail on Day 3. But right now, it is important for you to understand that an evasion at the TCP transport layer is very serious. What this means is that any traffic carried over TCP to one or more receiving host operating systems may not be scrutinized properly by the IDS/IPS, allowing malicious traffic to potentially remain undetected.



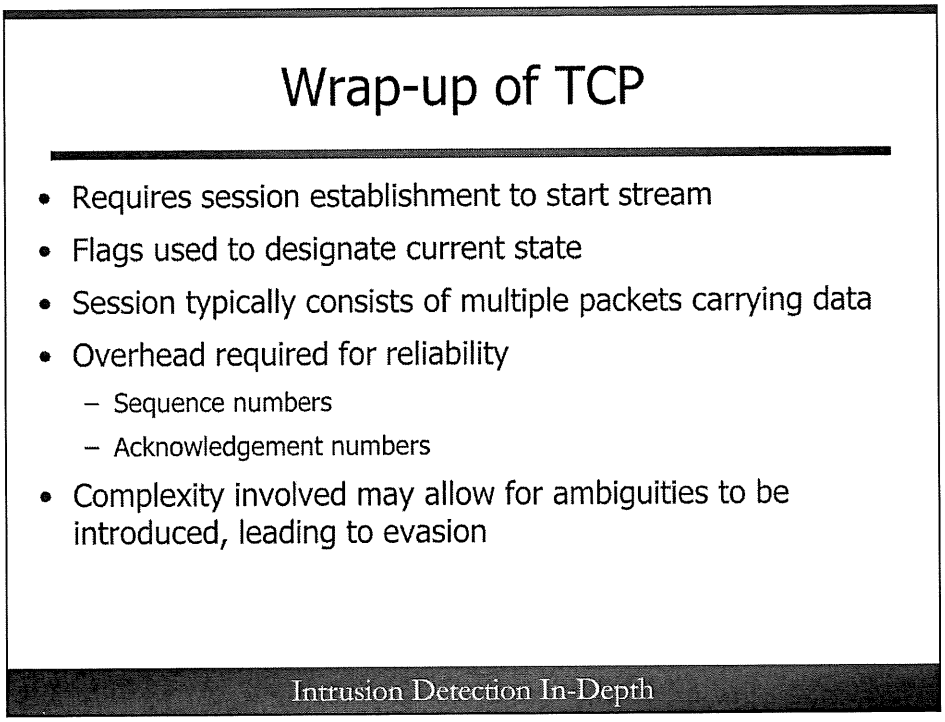
Let's look at a handful of TCP stream reassembly ambiguities. What should an IDS/IPS do with segments that belong to the same stream but have identical TCP sequence numbers, in other words, they overlap. Should the IDS/IPS honor the first or subsequent overlapping segment? And, whichever it chooses, will the destination host accept the same one? If the IDS or IPS makes the wrong choice and does not examine the same segment that the destination host receives, it is possible for it to miss an exploit.

While it is not an ambiguity, per se, how does an IDS/IPS deal with a TCP session where there is payload on a segment with wrapping TCP sequence numbers? This occurs when the TCP sequence number reaches the maximum value of $2^{32} - 1$ (the TCP sequence number is 32 bits long) and must wrap back to 0. This probably doesn't happen often, but an attacker can craft this type of traffic to see if an IDS/IPS handles it properly.

Other potential ambiguities exist that an attacker can use to confuse the issue of which segment to favor. For instance, what if another SYN comes along after a session has been established and the SYN appears to be for the same session yet it has a different TCP sequence number? Some operating systems will reject the new SYN and continue using the already-established session while others use the new SYN and session. This may be dependent on the value of the TCP sequence number found in the second SYN.

What about data found in the payload of a SYN segment? Most operating systems simply ignore it, but others buffer it pending the completion of the three-way handshake. How should an IDS/IPS handle segments that have superfluous flags or segments with payload that don't have both or either the PUSH or ACK flags? Again, different operating systems respond uniquely to unexpected combinations of TCP flags.

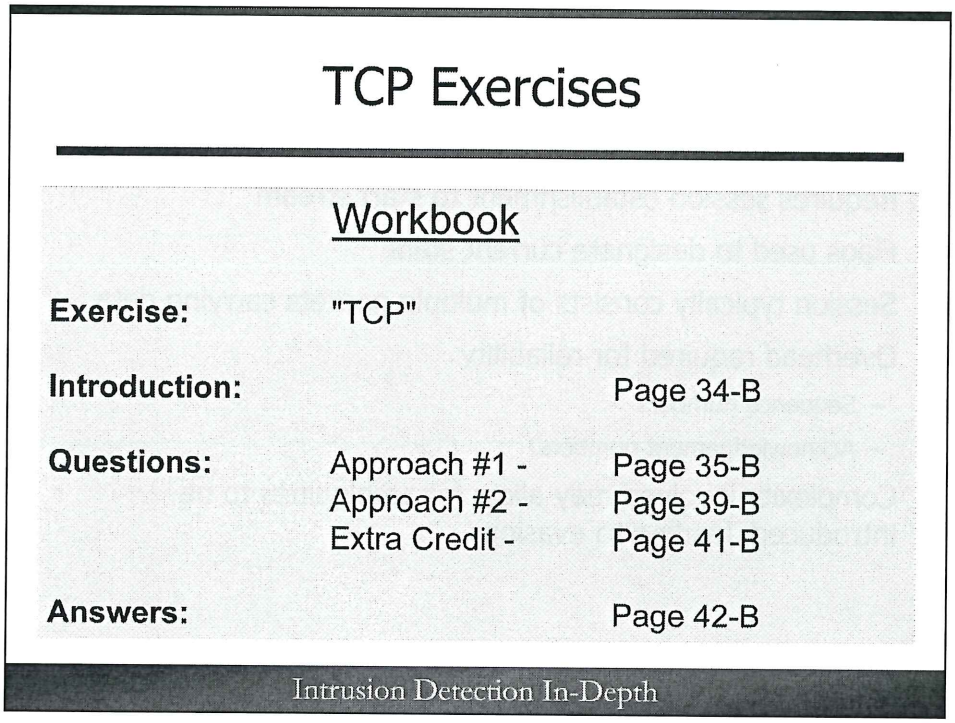
Finally, how about if an IDS/IPS does not see activity from a particular session after a certain amount of time – should it stop tracking the session? If the IDS/IPS does not stop tracking, it can become overloaded with dead sessions. If it stops tracking prematurely, it may miss alerting or blocking on malicious content.



In review, 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 can be exchanged after the three-way handshake. A typical 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, such as Linux acknowledging data sent in a segment with no TCP flags. 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.



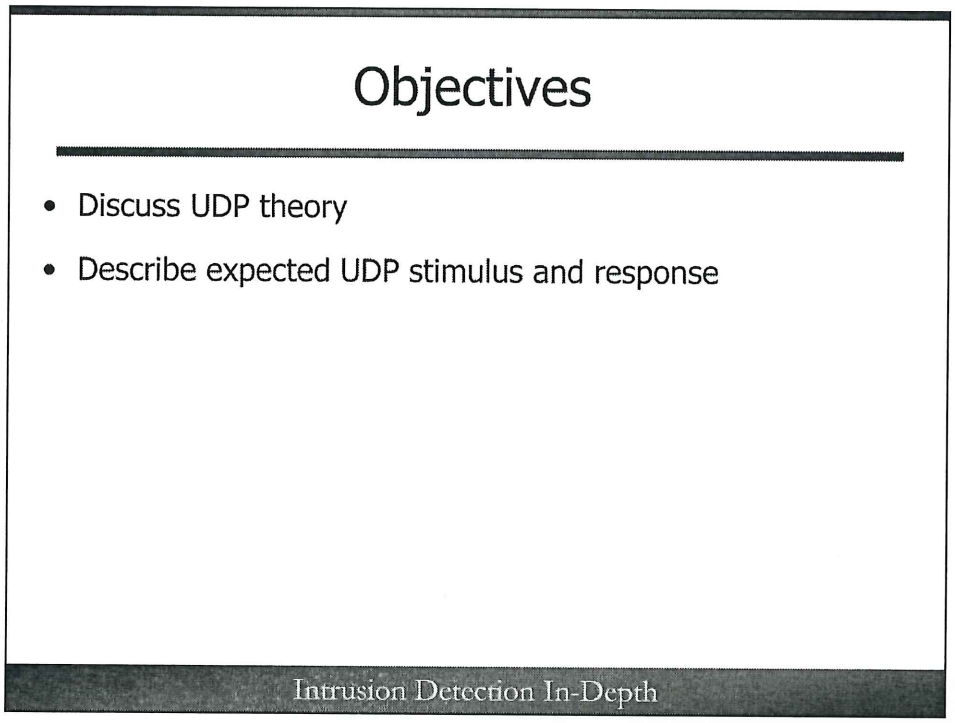
This page intentionally left blank.

UDP

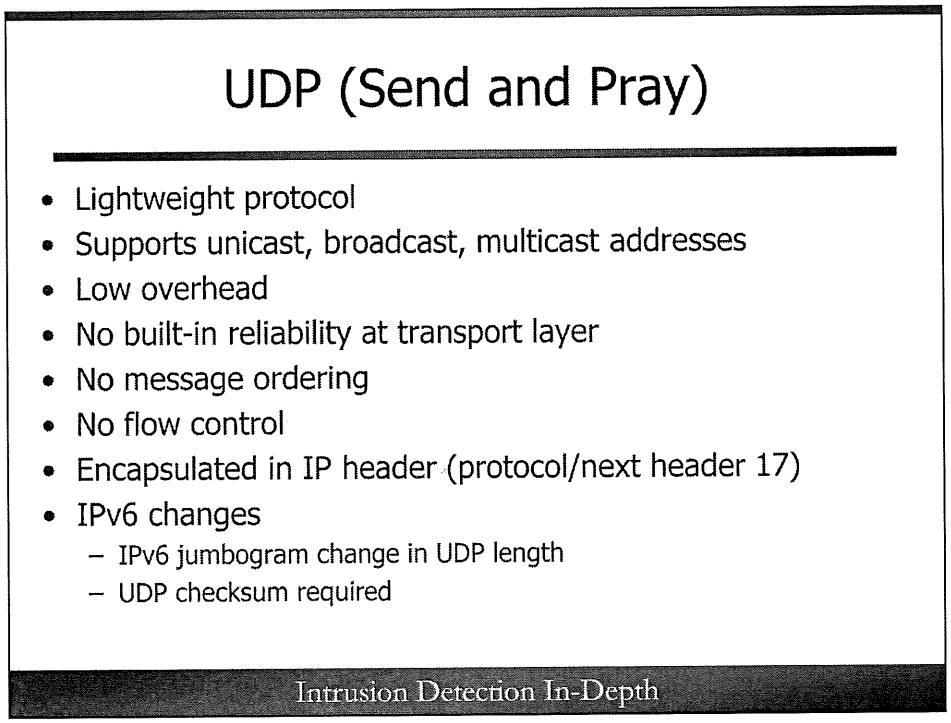
- Wireshark Display Filters
- Writing tcpdump Filters
- TCP
- UDP
- ICMP

Intrusion Detection In-Depth

This page intentionally left blank.



Compared to TCP, UDP is a simple protocol. It is often explained as a "send and pray" protocol since there is no guarantee of reliability. It has a simple header of 4 fields. We'll cover a couple of examples of sending UDP packets under different circumstances to understand the expected responses.



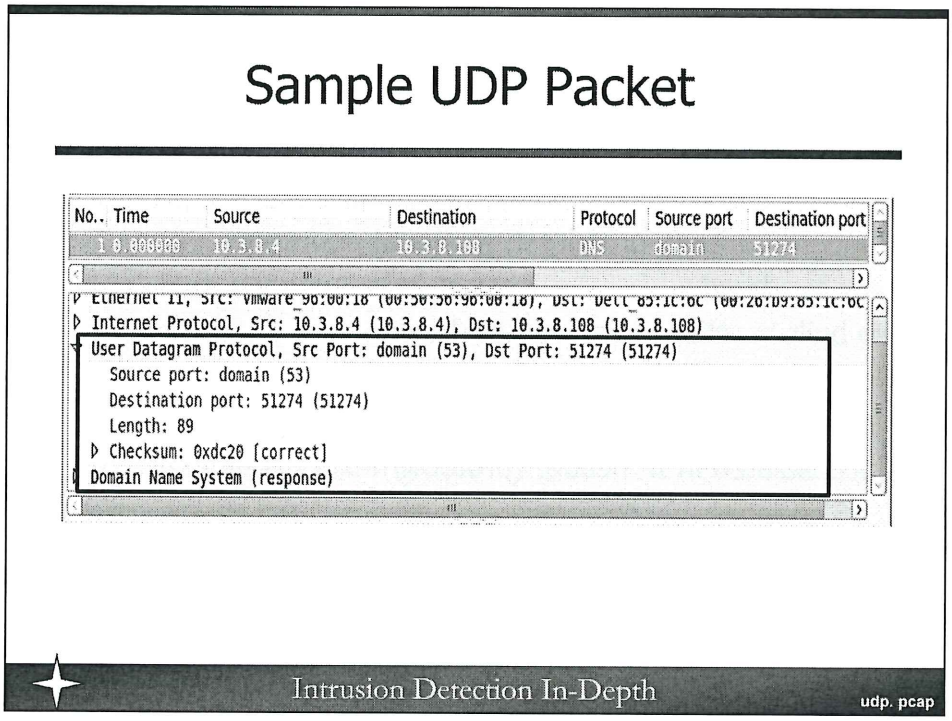
UDP is considered a lightweight transport protocol because it is simple and makes no guarantee of reliable delivery. Unlike TCP, that supports only single host to single host (unicast) communications, UDP can deliver traffic to one or more hosts. It has low overhead and can be very fast because it has a standard 8 byte header that carries vital information like 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.

As we would expect, UDP requires an IP header just like all other protocols. If the protocol/next header field of the IP header contains a value of 0x11 or decimal translation of 17, then you can expect a UDP header and data to follow the IP header.

UDP, like TCP uses source and destination ports. The destination port is the desired listening service or protocol, like DNS. The source port is an ephemeral one that the connecting host selects upon opening the connection.

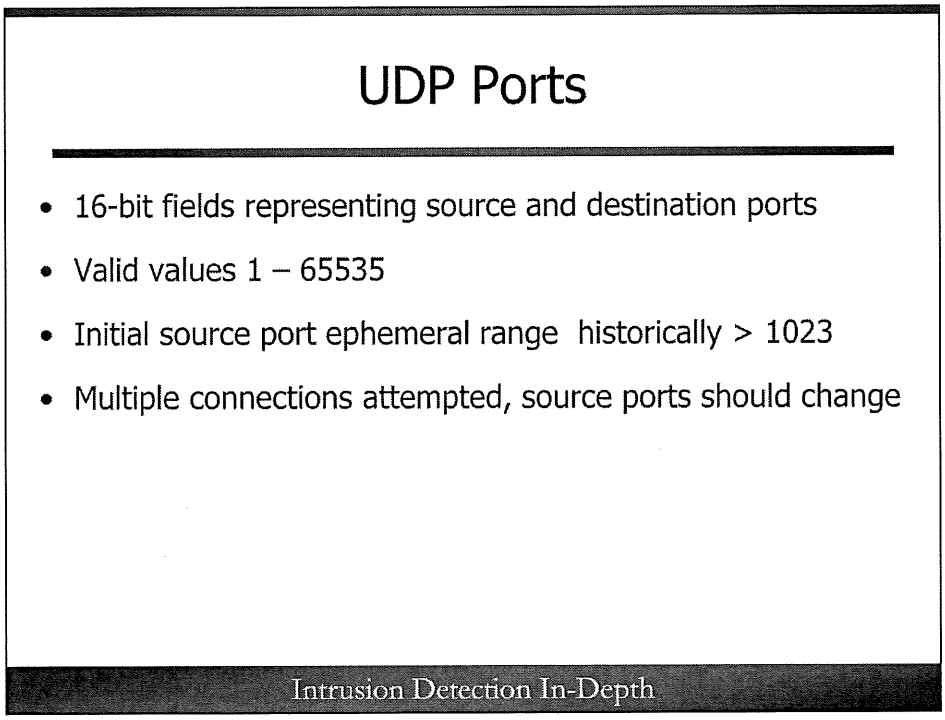
Like TCP, there must be some way to indicate that a packet is an IPv6 jumbogram. Because the IP length field cannot represent a value greater than 65535 another field must be used. A UDP length of 0 designates that the packet is a jumbogram. The UDP length field found in the UDP header reflects the number 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 good jumbogram indicator.



There really isn't much to a UDP header. The one shown above in Wireshark reflects a DNS response. Like TCP, 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 that the datagram is delivered to the correct host.

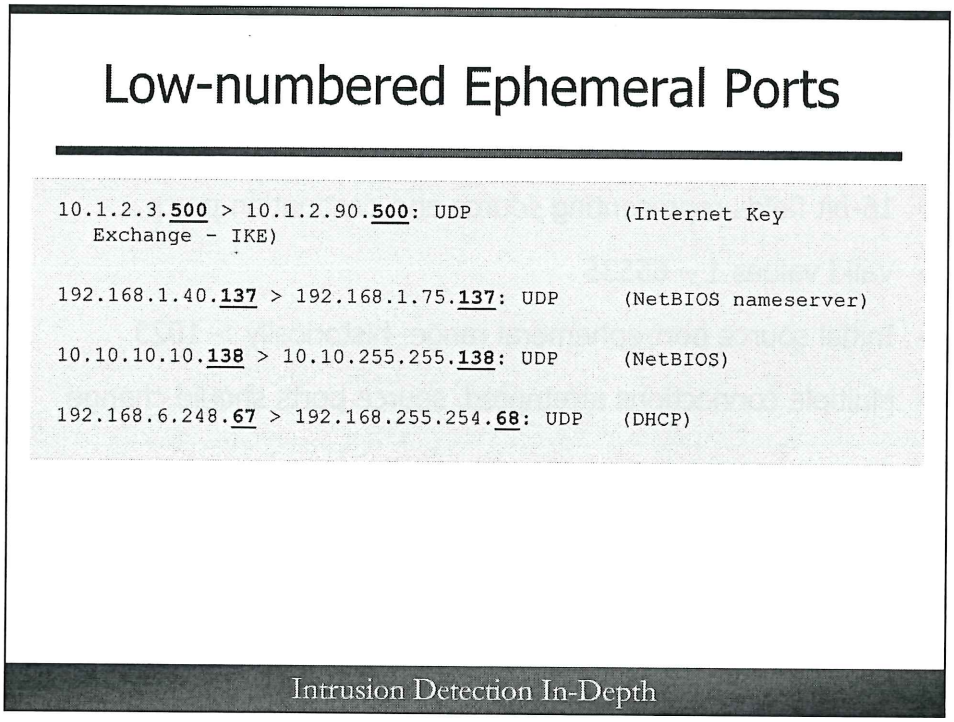
To display the output, enter the following command:

wireshark udp.pcap



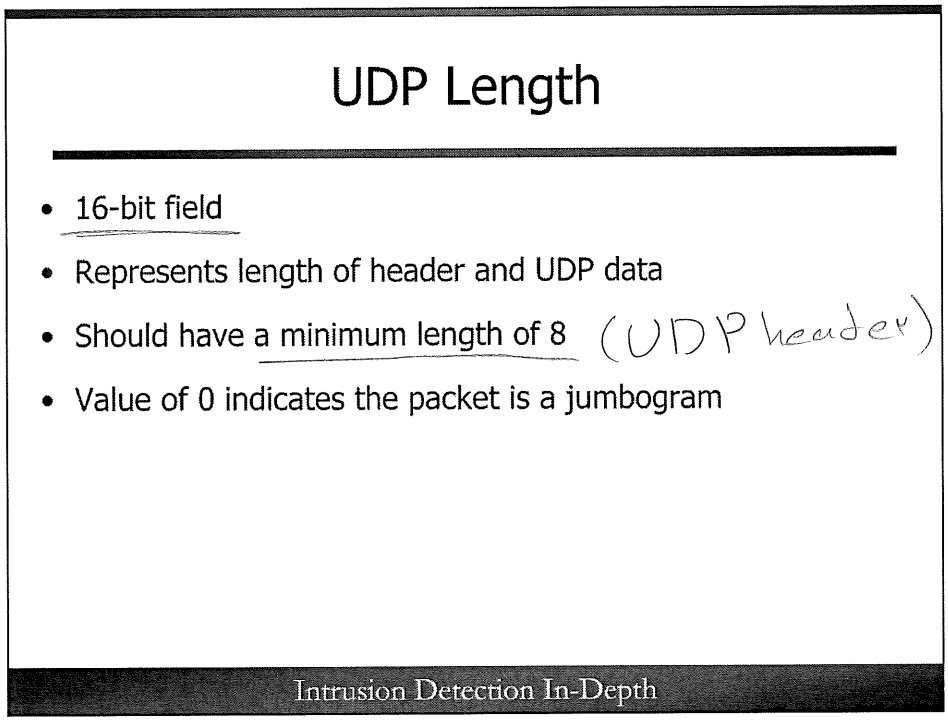
Just as with TCP ports, UDP port fields are two separate 16-bit fields in the TCP header, one for source and another for destination ports. The valid range of values is between 1 and 65535; the use of port 0 is typically not used in normal traffic.

Historically, when a source host wishes to connect to a destination host, an ephemeral port is typically selected in the range of ports greater than 1023. UDP source ports for more current protocols may not follow this convention. For each new sending connection, a different ephemeral port should be selected.

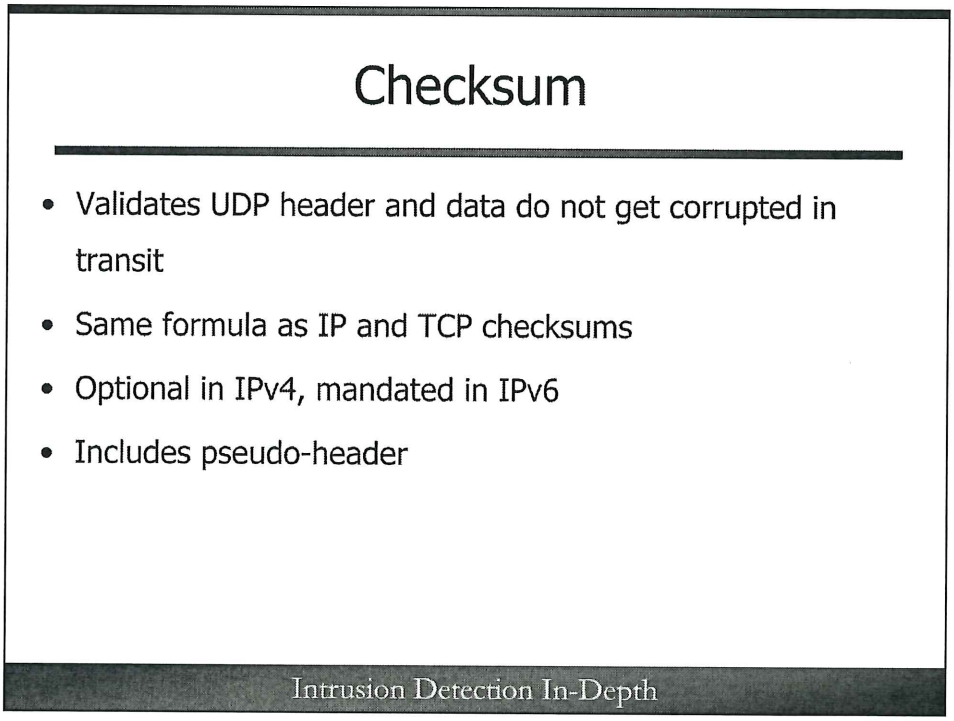


As we've learned, most clients connect to servers using ephemeral ports historically in the range of 1024 and above. Yet, we see that is not always the case. If you recall, historically, the ports below 1024 were reserved for well-known server ports.

However, there are some "protocol-benders" that use client port numbers that go against convention. In the first three records of tcpdump output, all the protocols displayed use the same client and server port numbers. Some of the protocols that do this are Internet Key Exchange on port 500, NetBIOS nameserver port 137, and NetBIOS port 138. The fourth record represents a client issuing a Dynamic Host Configuration Protocol (DHCP) request for an IP address from ephemeral port 67 to server port 68.

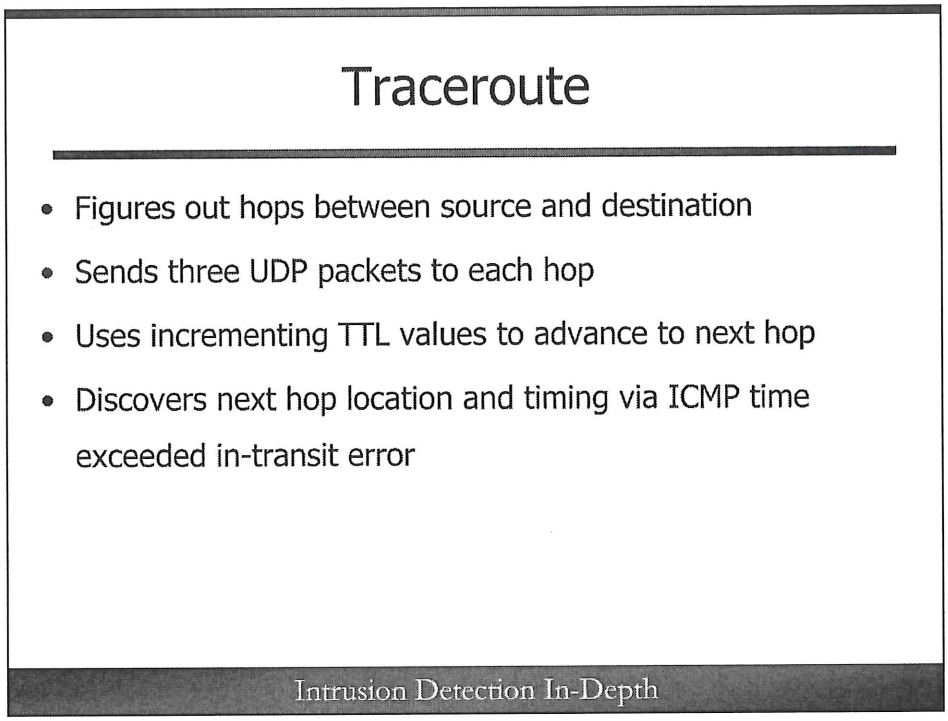


The UDP length is the number of bytes found in the UDP header plus the number of bytes found in the payload. The UDP header is 8 bytes so the minimum length for the UDP length is 8 bytes. Remember that a 0 in this field is the indicator that the packet is a jumbogram.



Like TCP, the UDP header has a checksum field. This checksum value is computed by taking the standard 16-bit one'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 datagram 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 that the datagram arrives at the correct host.

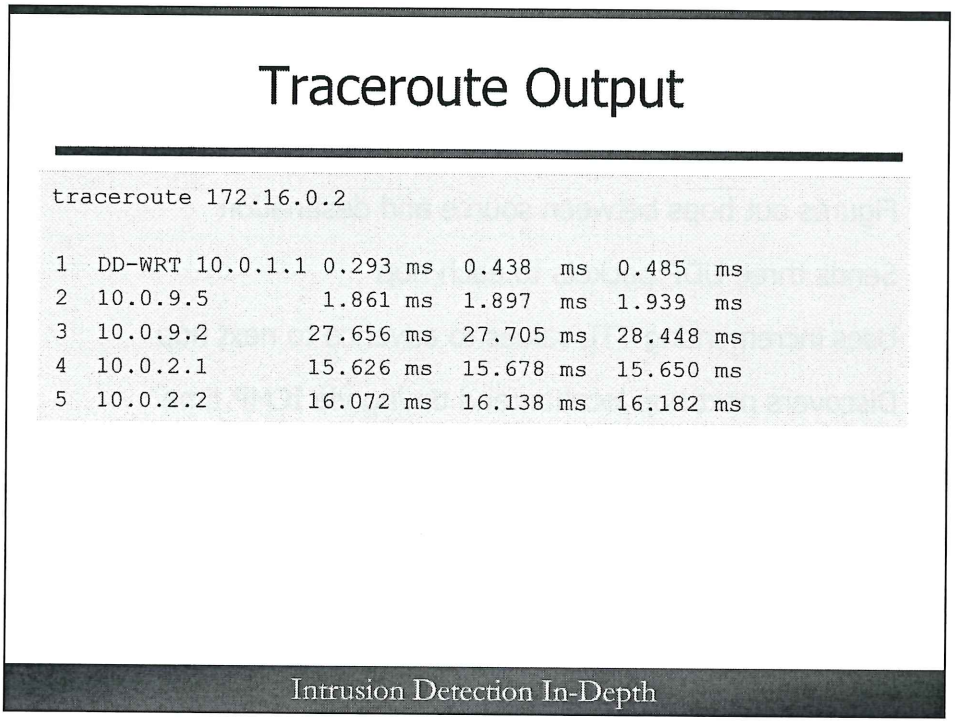
The UDP checksum is optional in IPv4, though almost always used. It is mandated in IPv6 since there is no IP header checksum, causing the transport layer to include a checksum.



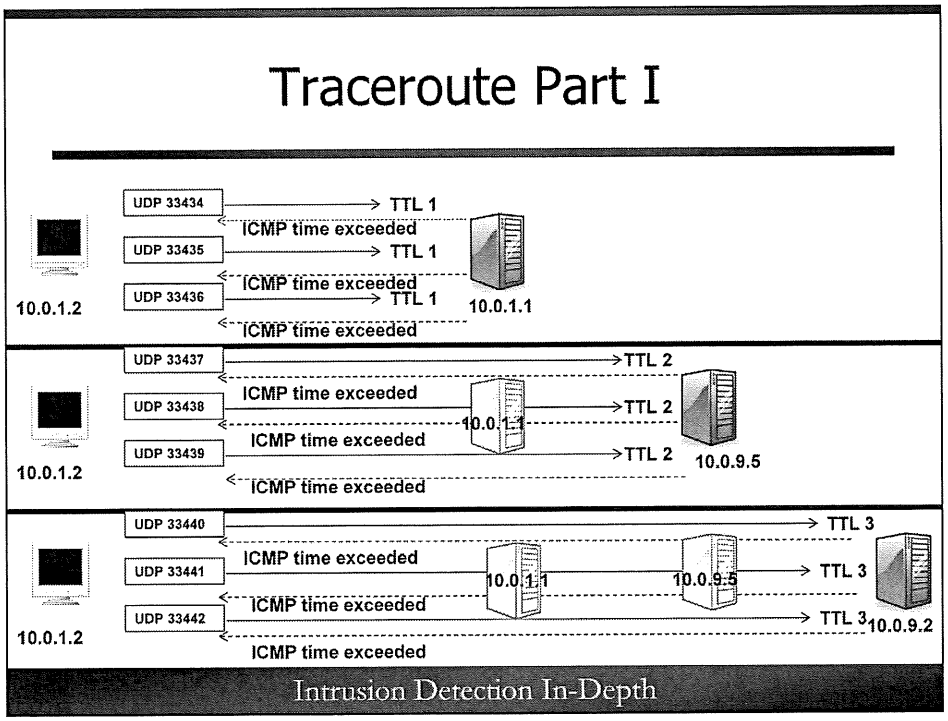
The Unix traceroute command is a combination of UDP and ICMP used to discover the path that a packet takes from source to destination. This traceroute program is similar in function to the Window's tracert, however instead of using ICMP to discover the routers and destination host, traceroute uses UDP.

Traceroute uses UDP to send three packets with a destination port of 33434-33534 to the next hop. Three packets are sent to each hop for redundancy in case there is packet loss. Each of these three packets has a TTL of 1 when sent to the first hop. As you would expect, the first hop device returns an ICMP "time exceeded in-transit" message because the decremented TTL value would be 0. The traceroute program on the sending host records this as the first route. Each subsequent hop from source to destination receives three UDP packets with incrementing UDP ports and a TTL value of one more than the previous one. Each of the hop devices returns the ICMP "time exceeded in-transit" error and again traceroute records this 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.

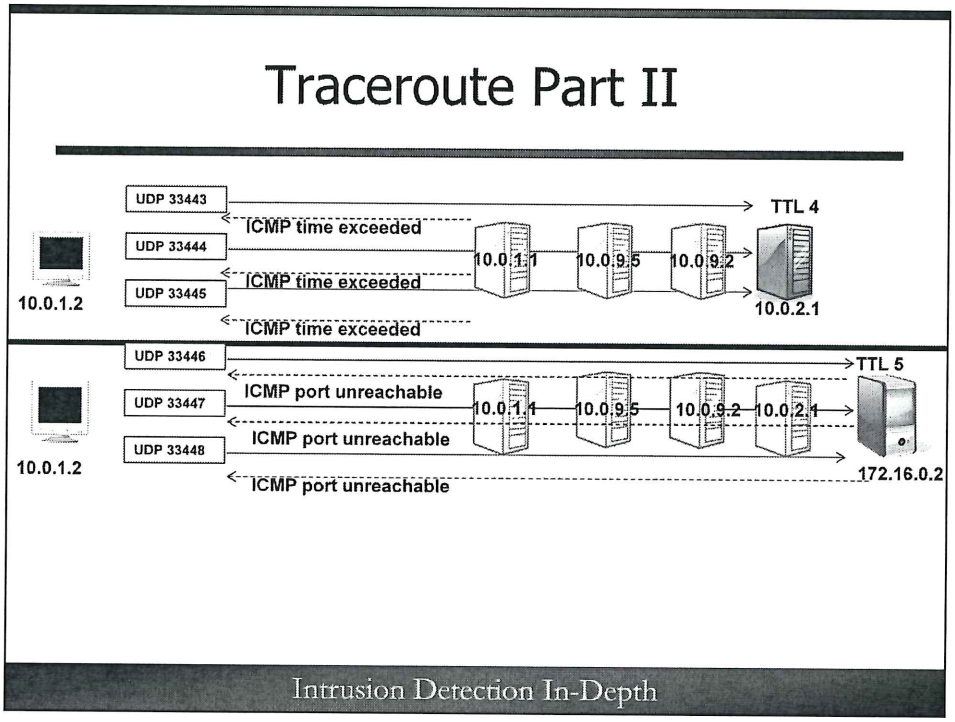


Let's say that host 10.0.1.2 performs a traceroute to 172.16.0.2 Here is the type of output you might see. Traceroute reports each hop as it is returned in an ICMP "time exceeded in-transit" message. The first set of three UDP packets is sent to hop 10.0.1.1; all have a TTL of 1; and each of the ICMP error messages is returned whereupon traceroute records the round-trip time of each. Next, three more UDP packets are sent to the next hop of 10.0.9.5 with a TTL of 2. These packets get routed past the first hop of 10.0.1.1 and each is returned with an ICMP "time exceeded in-transit" message. This continues through hops 10.0.9.2, 10.0.2.1 and 10.0.2.2 where each set of UDP packets gets an incremented TTL value per hop and each individual UDP packet gets an incrementing destination port.

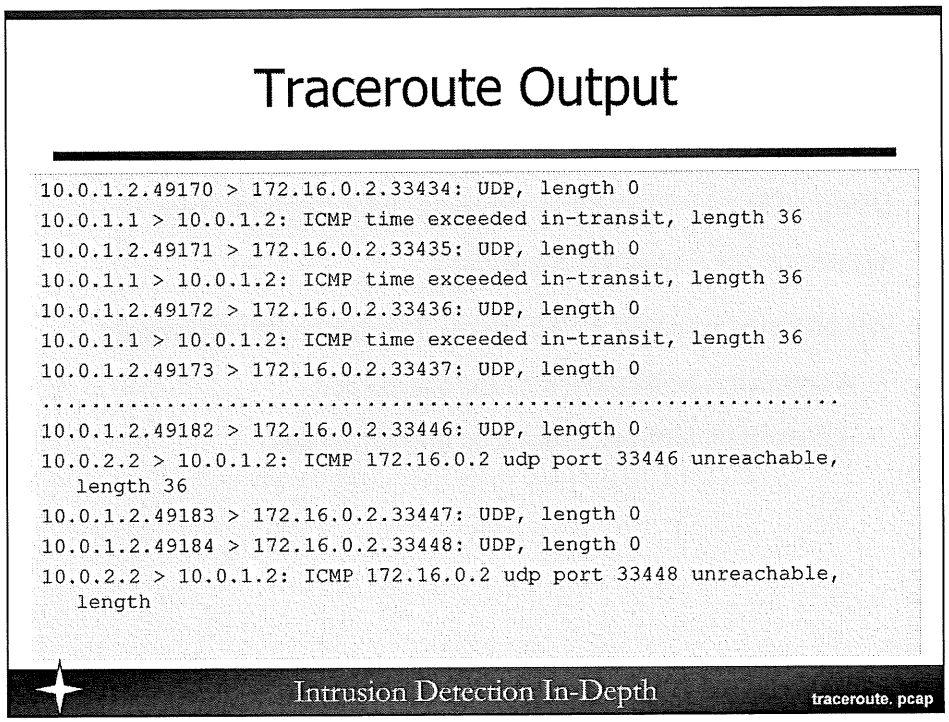


Perhaps it might be easier to understand traceroute by looking at a pictorial representation of what is happening. At the top, you see the initial discovery of the first hop. Three UDP packets are sent, each with a different UDP port of 33434, 33435, and 33436. Each has a TTL value of 1. And, 10.0.1.1 as the first hop returns 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, 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 334340, 33441, 33442, and each has a TTL value of 3.



The fourth hop of 10.0.2.1 is reached using a TTL value of 4 on each of the UDP packets destination for 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. Once the traceroute program receives this ICMP error, it knows that it has reached the target host.

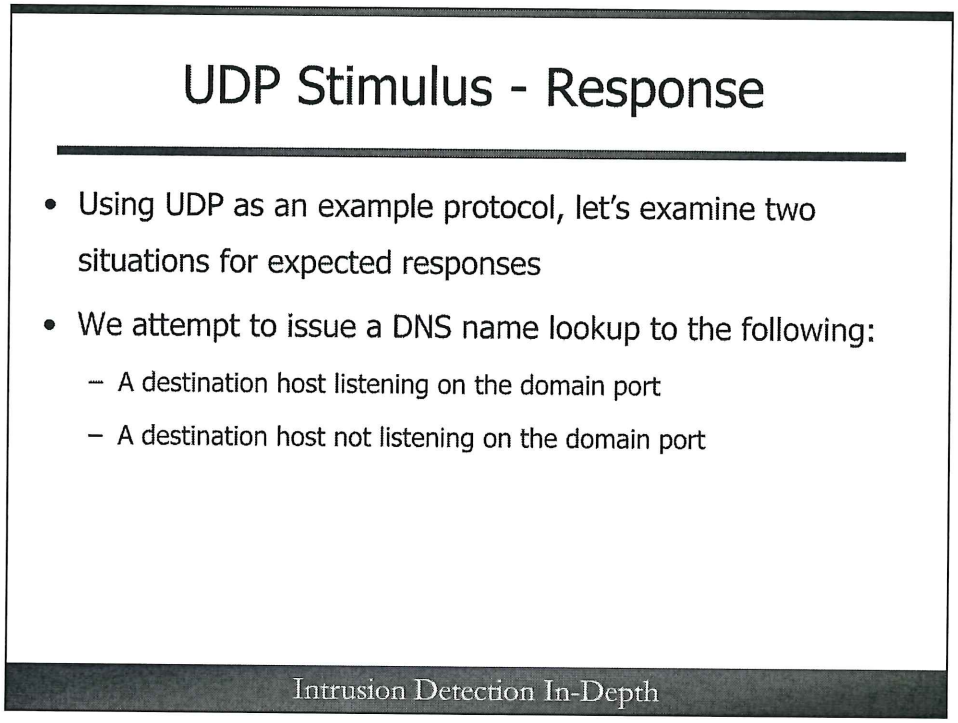


Here is the tcpdump output from the traceroute. The beginning and ending packets are shown. The pattern is that three UDP packets are sent with a TTL value to advance one hop. An ICMP "time exceeded in-transit" message is returned for each of these. The final five packets represent two UDP packets reaching the destination host of 172.16.0.2. It appears that either the UDP packet with destination port 33447 was never received or that ICMP unreachable message was never successfully returned.

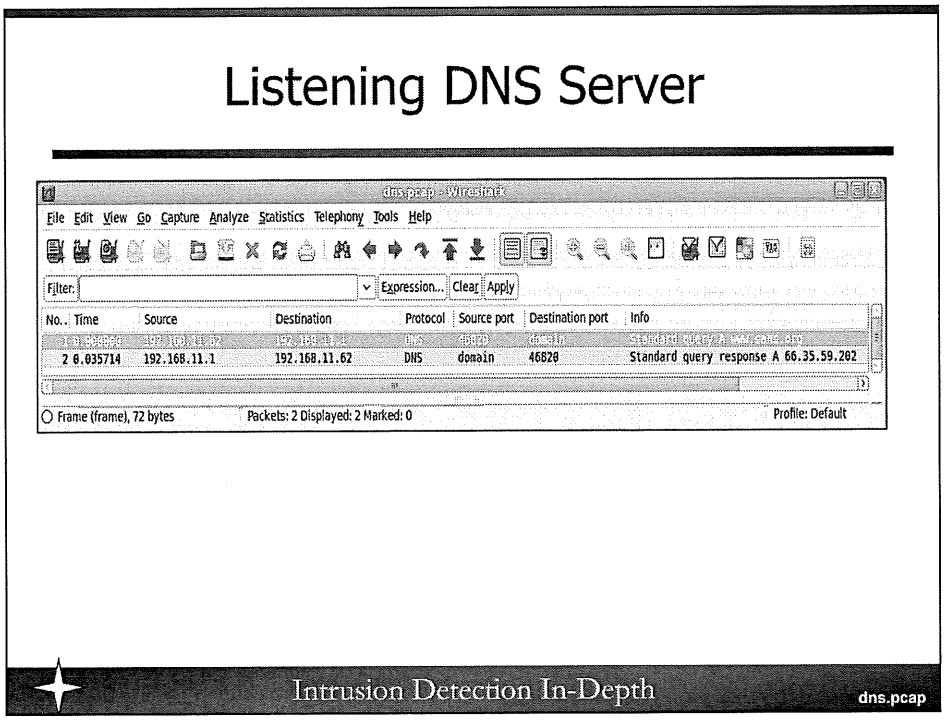
An oddity worth noting is that router 10.0.2.2 reports the 172.16.0.2 ICMP "port unreachable message". Usually, the host itself would report this. Perhaps 10.0.2.2 is a proxy for the 172.16.0.2 .

To display the output, enter the command:

tcpdump -r traceroute.pcap -nt

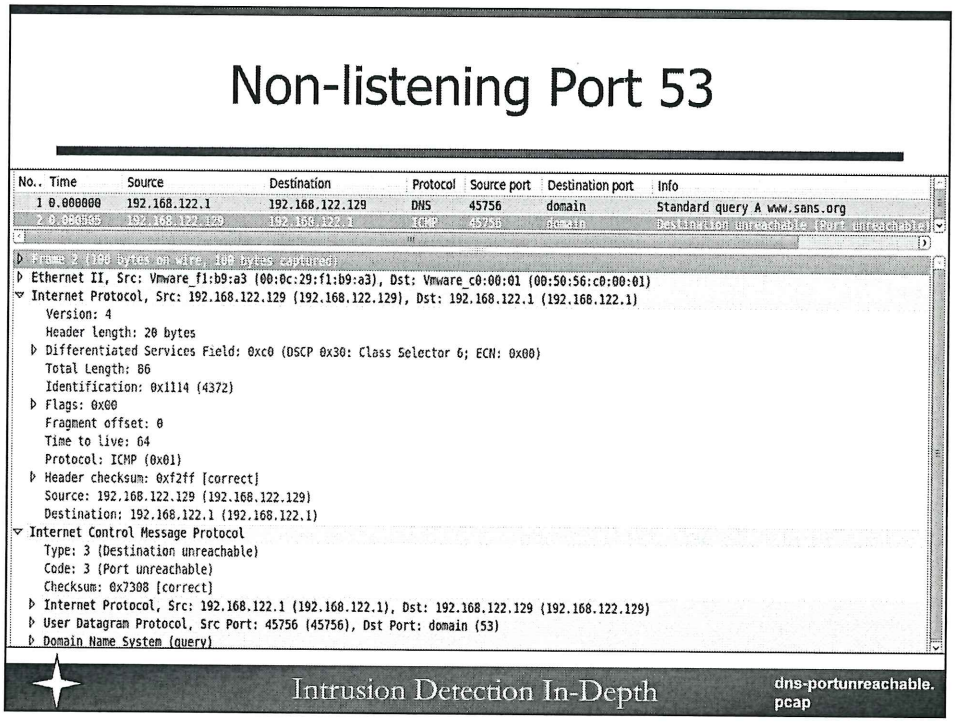


Let's look at how UDP responds to incoming packets. We use a DNS query to examine how UDP behaves. Specifically, we'll look at a listening domain port and a non-listening one. The other conditions examined for TCP such as a host that doesn't exist or the domain port blocked at the router will elicit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



This is Wireshark's interpretation of a DNS query of www.sans.org. The DNS server 192.168.11.1 listens on port 53 and returns a response. Be aware that this is not UDP responding, but the application – in this case DNS. That is true for all UDP ports and applications – responses are received from the application only.

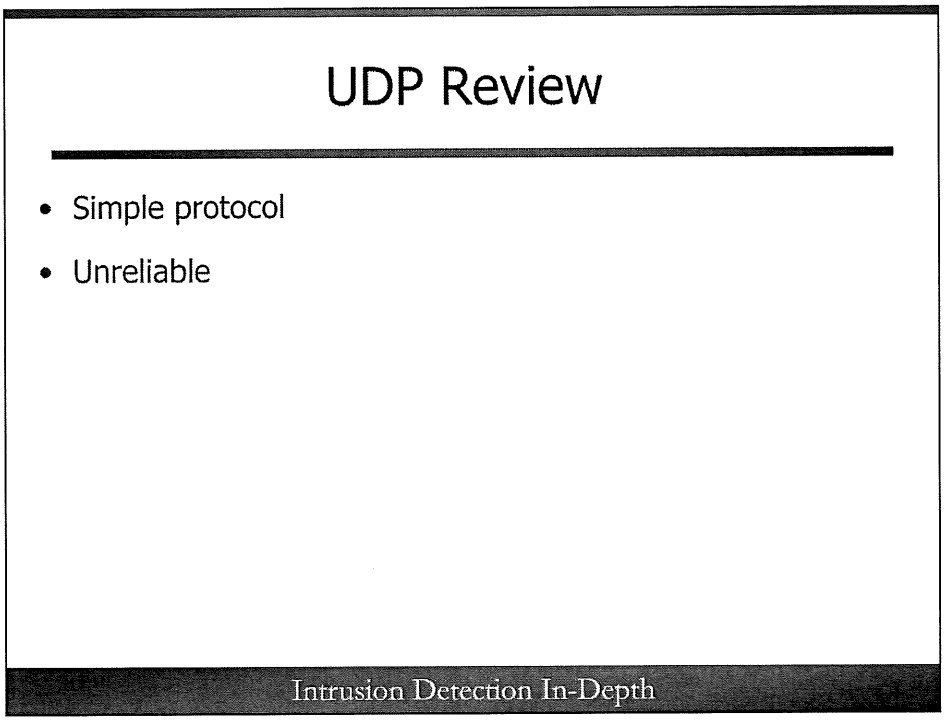
To display this output, enter the following on the command line:
wireshark dns.pcap



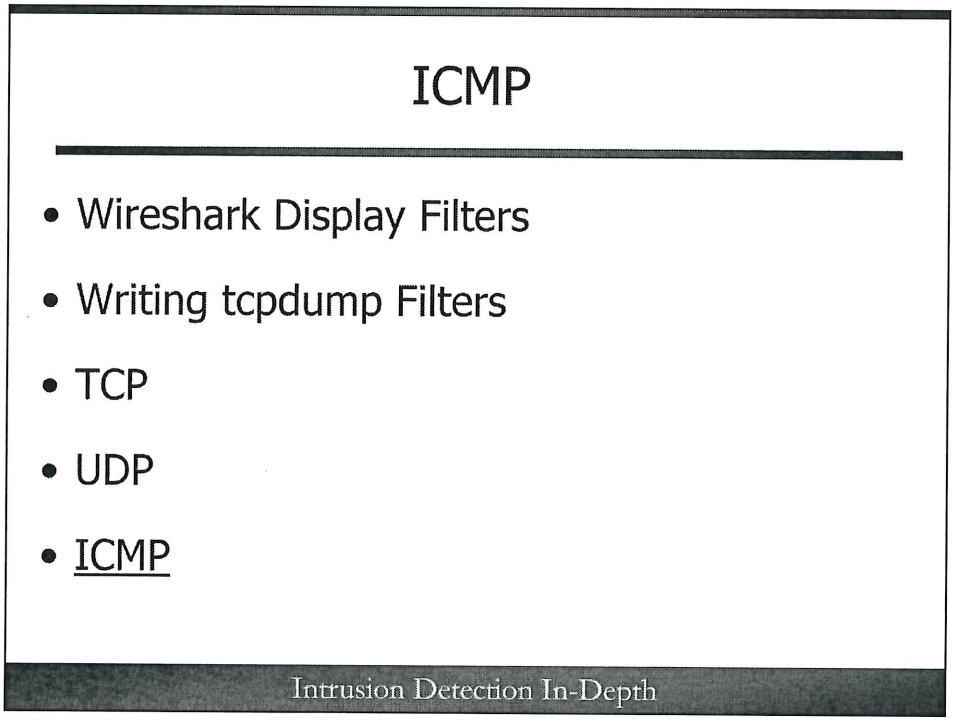
Now, let's observe the response when the domain port is not active. In this case, 192.168.122.129 responds that port 53 is unreachable.

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



There really is not much to UDP. It's a simple protocol most often used for short queries and responses such as DNS. It is inherently unreliable meaning that an application that runs over UDP that requires reliability must have some mechanism to ensure packet delivery.



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. It is important for you to understand how this protocol is used for both its intended purpose as well as for malicious purposes.

Some of the theory that you will learn from in this section is where ICMP fits in with other protocols such as IP, TCP, and UDP. You will also learn to understand the difference in ICMP and the other protocols. We will follow a familiar format we've seen in many of the other sections of looking at the conventional ICMP traffic as well as examining how ICMP can be used for nefarious activity.

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.

If you are interested in reading more about ICMPv4, see RFC 792, for ICMPv6, look at RFC 2463.

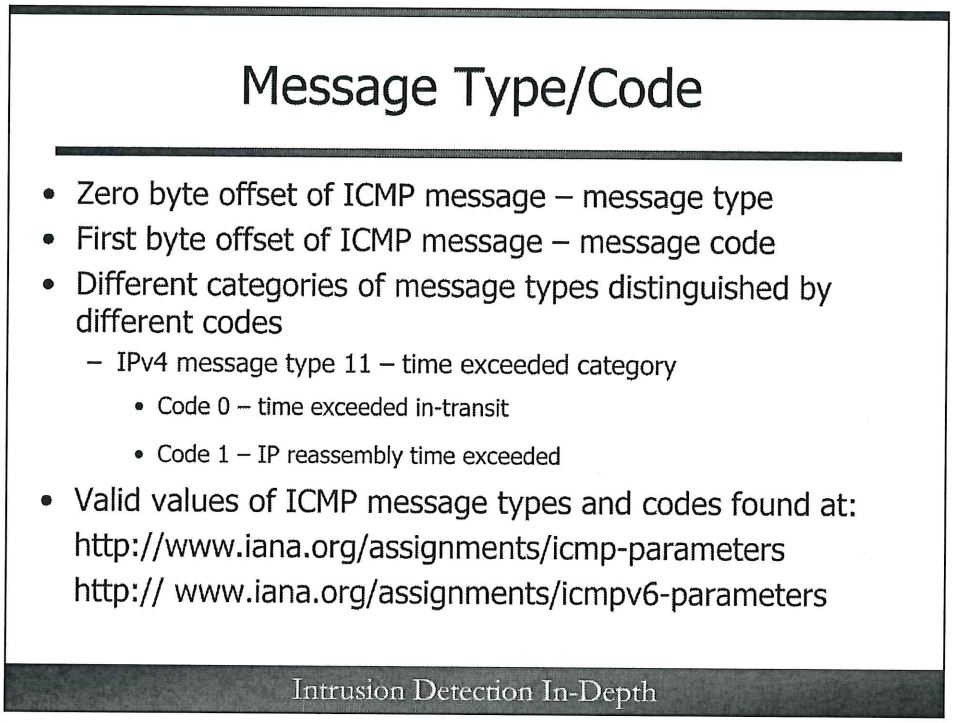
Objectives

- Discuss why ICMP is needed, where it fits in, and how it is different from the other protocols
  - Explain the theory for the above topics
- Examine how ICMP is used to map networks
  - Show examples using tcpdump output
- Examine classic ICMP activity
  - Show examples using tcpdump output
- Examine malicious ICMP activity
  - Show examples using tcpdump output

Intrusion Detection In-Depth

We'll cover several aspects of ICMP traffic in this section. ICMP has a special place among the protocols; it is unlike TCP and UDP.

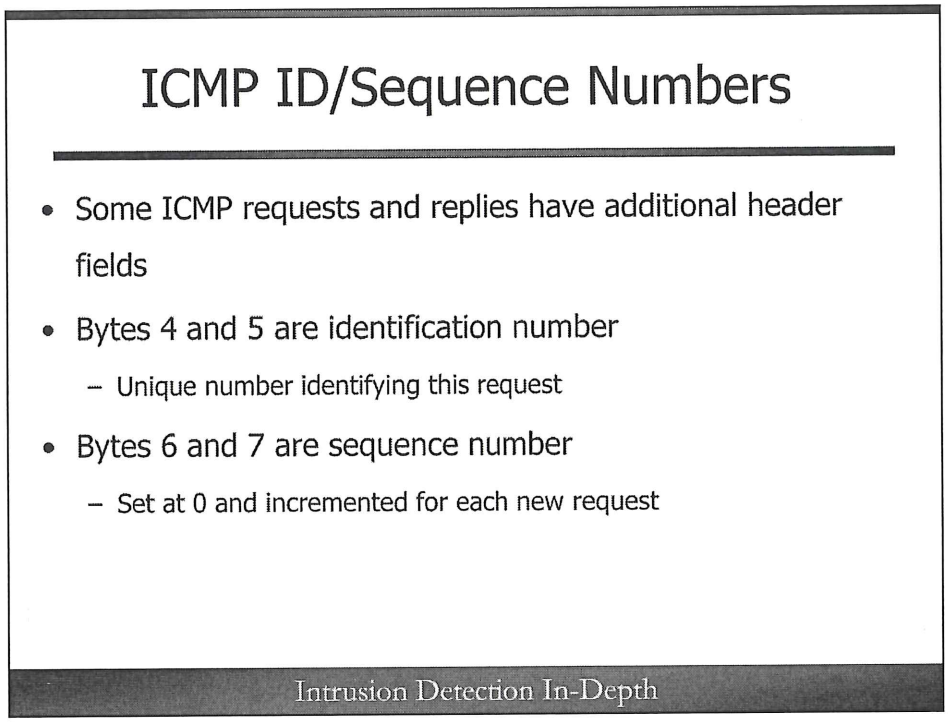
We'll take a look at how ICMP is used to map a given network, often as part of the reconnaissance phase to prepare for some kind of additional activity such as a scan. Next, we'll examine the expected behavior that ICMP exhibits. And finally, we'll look at the many ways that ICMP has been mutated and tainted to perform activity that it never was meant to perform.



Remember ICMP has no ports. So there has to be a method of uniquely indicating what type of ICMP message is being sent or received. The first two bytes of the ICMP message are the ICMP message type and code. The message code is a subcategory under the message type.

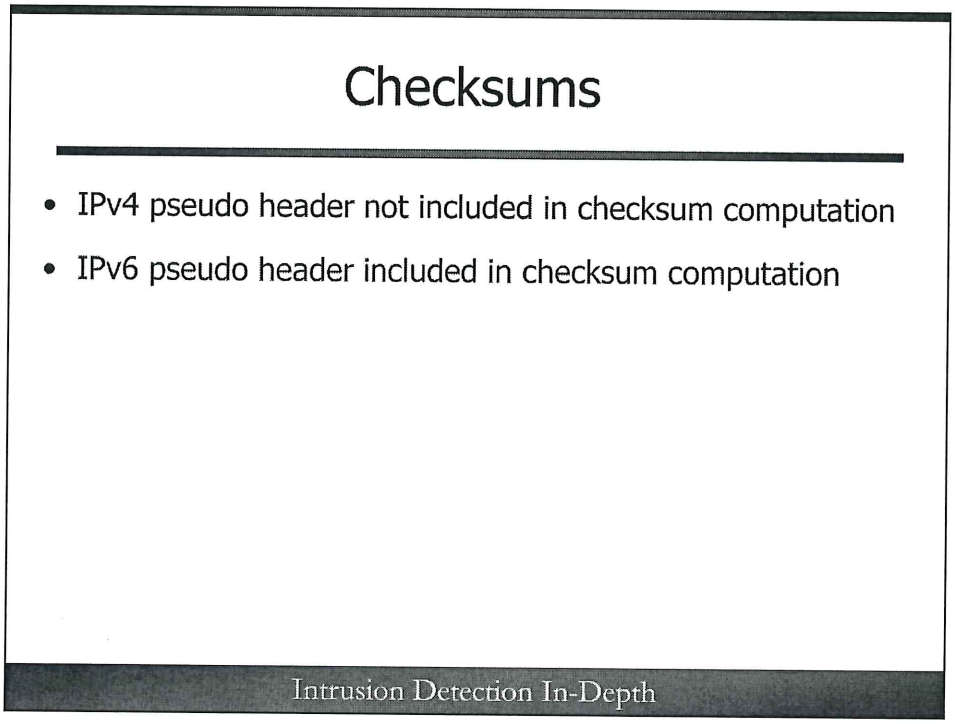
For instance, there are two possible message codes for a message IPv4 type of 11 which represents the time exceeded category. If the message code is 0, it is a "time exceeded in-transit" message. If the message code is 1, it is an "IP reassembly time exceeded" message.

ICMP has changed in IPv6 including the type and code values for ICMP messages. For instance, ICMP type 128 code 0 is now an echo request and ICMP type 129 code 0 is an echo reply. Recall that in IPv4 an echo request was type 8 code 0 and an echo reply was type 0 code 0. There are different types of ICMP messages in IPv6 as well.

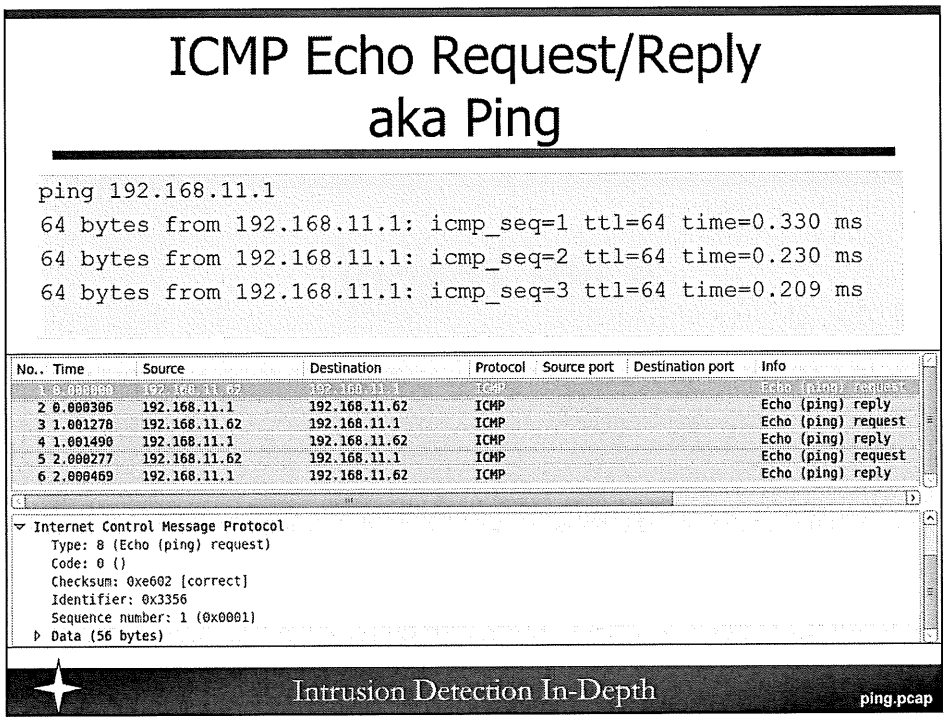


If you examine ICMP echo requests, you'll find some additional fields in the ICMP header. These are the ICMP identifier found in bytes 4 and 5 offset and the ICMP sequence number found in bytes 6 and 7 offset.

These fields are used in an echo request/reply pair to uniquely identify requests and pair them with responses and keep track of the replies. For Unix hosts, the ICMP ID is typically the process ID number of the ping that generated the traffic. There can be several simultaneous ping commands so the identifier in both the echo request and reply will inform the pinging host the associated echo reply for the issued request. Each ping can generate several echo requests and the sequence number is the manner in which they are tracked and paired with the reply.



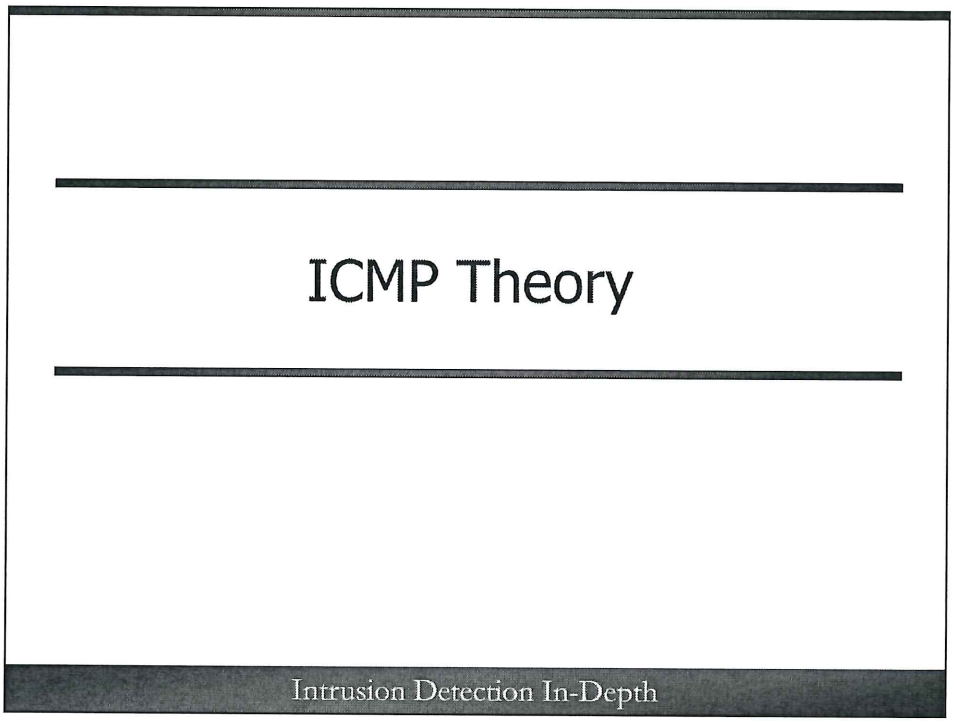
The ICMPv4 checksum uses the ICMP header and message to compute the checksum. The pseudo header, including fields from the IPv4 header, wasn't deemed necessary for ICMP checksum since the IPv4 header had its own checksum as a cross check for the ICMP checksum. But, in ICMPv6 there is no IPv6 checksum, necessitating the pseudo header computation that includes values from the IPv6 header.



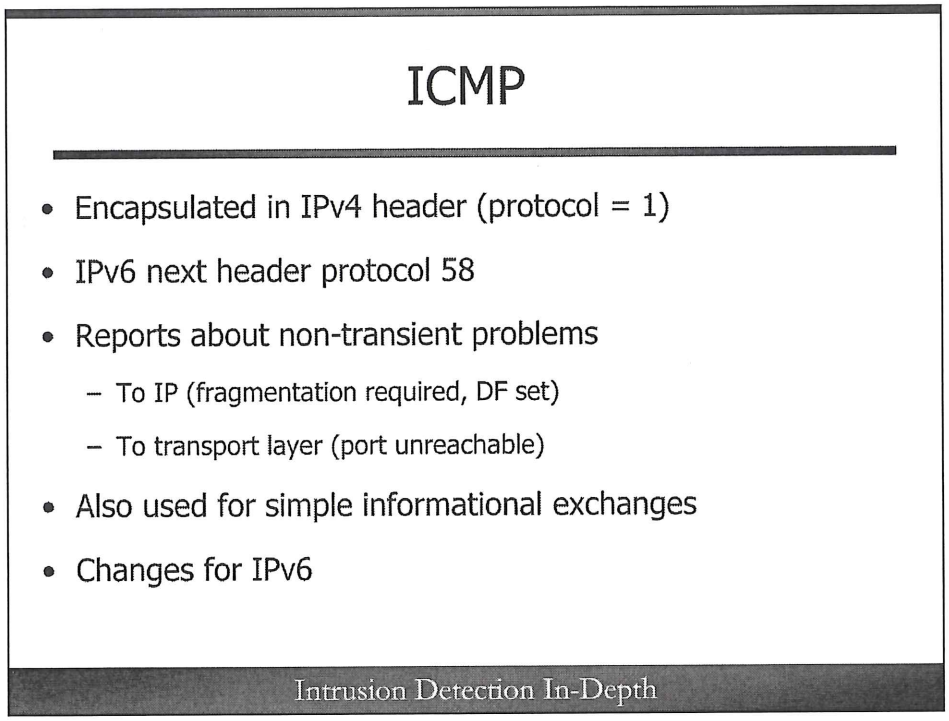
This is a set of three echo requests and replies. The first echo request has an ICMP identifier of 0x3356 and a sequence number of 1. The two other echo requests share the same ICMP identifier with a sequence number that increments by 1.

To view this output, enter the command:

wireshark ping.pcap



We'll attempt to understand the need for ICMP, how it compares with other protocols and how it supports other protocols. We learn that ICMP has a unique purpose and a unique method of communication.

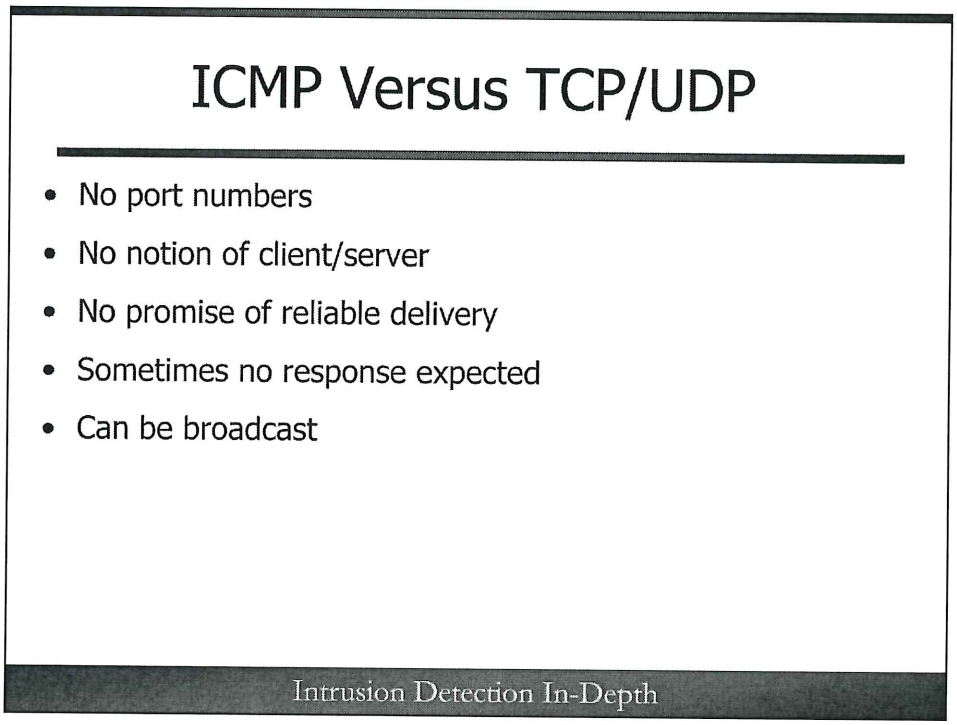


ICMP was created to report about problems that are not transient or likely to go away soon. For instance, if a packet is sent and becomes corrupted in transit, this error is not reported by ICMP. Data corruption is considered to be transient and will likely disappear the next time the packet is sent - 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 a 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 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.

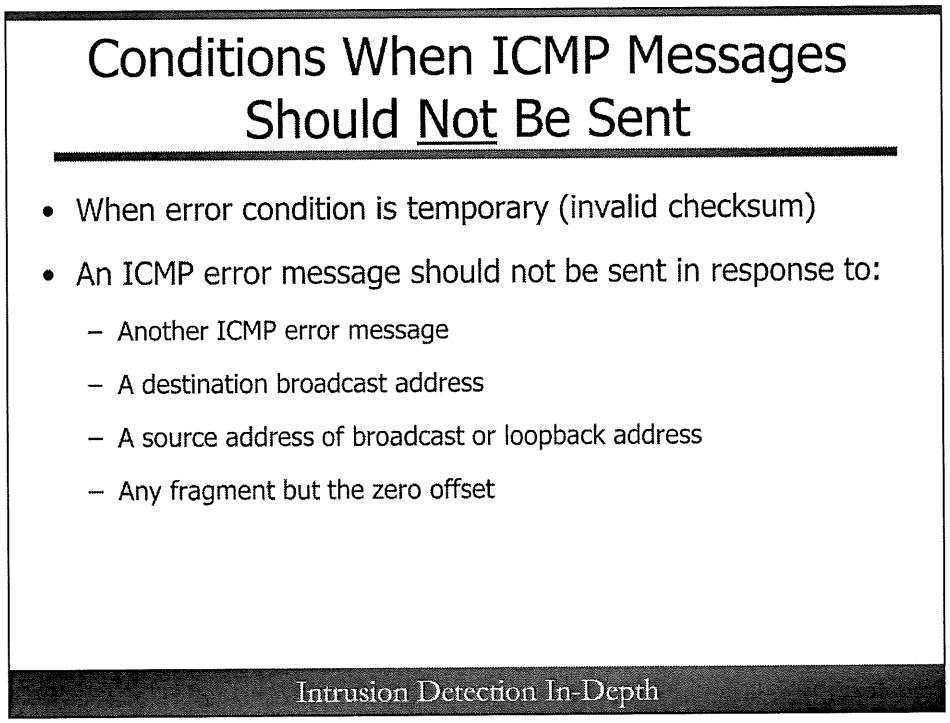
ICMP messages are used for simple informational exchanges as well. The most well-known ICMP application is certainly the ICMP echo request/echo reply, or ping. Ping is used to find whether a given Internet host is reachable or not. ICMPv6 has greatly extended the use of ICMP to include network discovery via Neighbor Discovery Protocol. NDP messages can be considered informational exchanges too.



Moving ahead, we see that ICMP is different than TCP and UDP in other ways. The first is that ICMP has no port numbers. ICMP is able to differentiate functions using an ICMP message type and code, the first two bytes in the ICMP header. These tell the function of the particular ICMP message.

Next, there is really no such thing as a client and server.

Another trait about ICMP is that it supports broadcast traffic. TCP required an exclusive client-server relationship, but ICMP isn't nearly as possessive. We'll see where this flexibility in ICMP's ability to send and respond to broadcast messages can sometimes cause problems.



Okay, now that we have a handle on ICMP messages and when they are sent, it is almost as important to know the conditions when ICMP messages should not be sent. This will give us a more comprehensive view of ICMP.

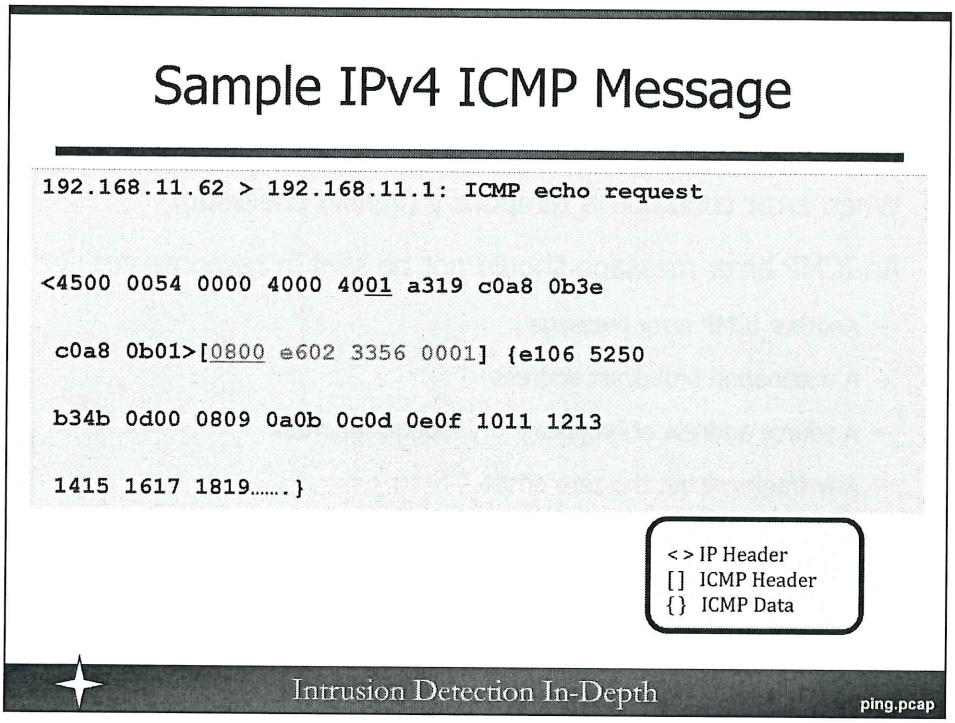
We mentioned on the previous slide that ICMP messages should be sent for non-transient conditions only. In other words, if the condition is temporary and likely to be corrected without an error message, ICMP messages will not be sent.

ICMP messages should not be sent in response to receipt of another ICMP error message. For instance, let's say someone spoofs a packet from an existing source IP address and sends it to a non-listening destination port. The destination host will report that the port is unreachable and send 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 zero offset fragment in a fragment train should generate an ICMP error message. Generating multiple error messages for all fragments would send unnecessary traffic.

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



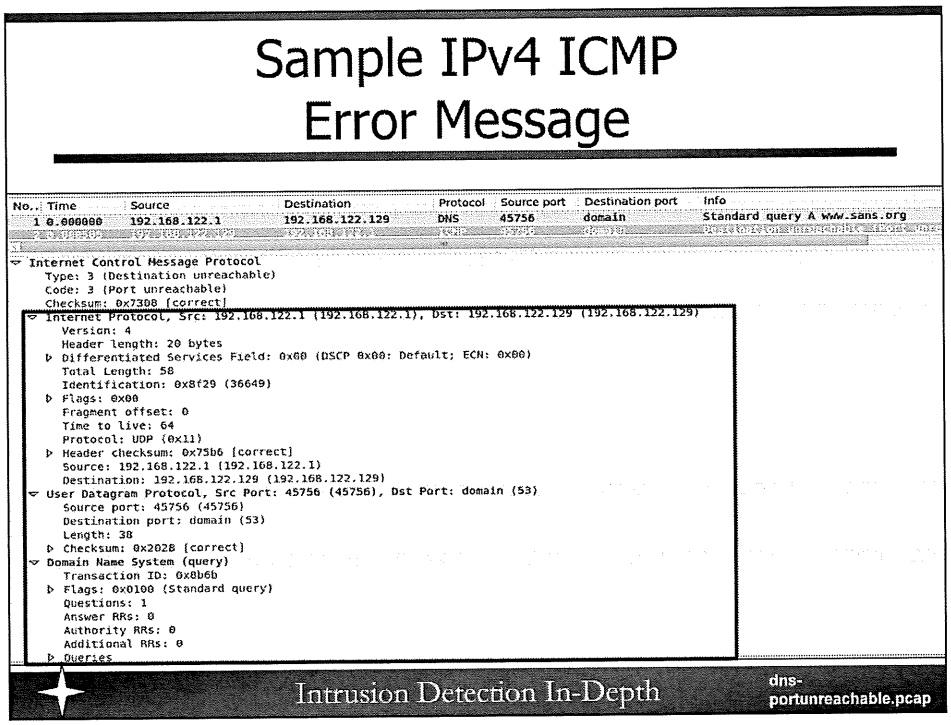
Use your hex packet ninja skills to break down the component parts of an ICMPv4 echo request. First, as always, there is an IP header found between the < >. As we've become accustomed to doing, we look at the 9th byte offset, which has been underlined in the IPv4 header, to see what the embedded protocol is. We discover that it is ICMPv4 because we find a 0x01 there. We already knew that from the standard tcpdump output, but it helps to validate this.

After the IPv4 header, we find the ICMP header in between the brackets. We see two bytes of underlined hex values of 0x0800. For ICMP, the first two bytes of the ICMP header contain two important fields – the ICMP type and the ICMP code. The 0x08 ICMP type and 0x00 ICMP code are what indicate that this is an ICMP echo request. If you are interested in examining the many combinations of ICMP types and codes, look at http://www.iana.org/assignments/icmp-parameters.

Finally, after the ICMP header, we find the ICMP data between the braces. ICMP echo requests may have different kinds of payloads depending on the sending host's implementation.

To display the output, enter in the command line:

tcpdump -r ping.pcap -nt

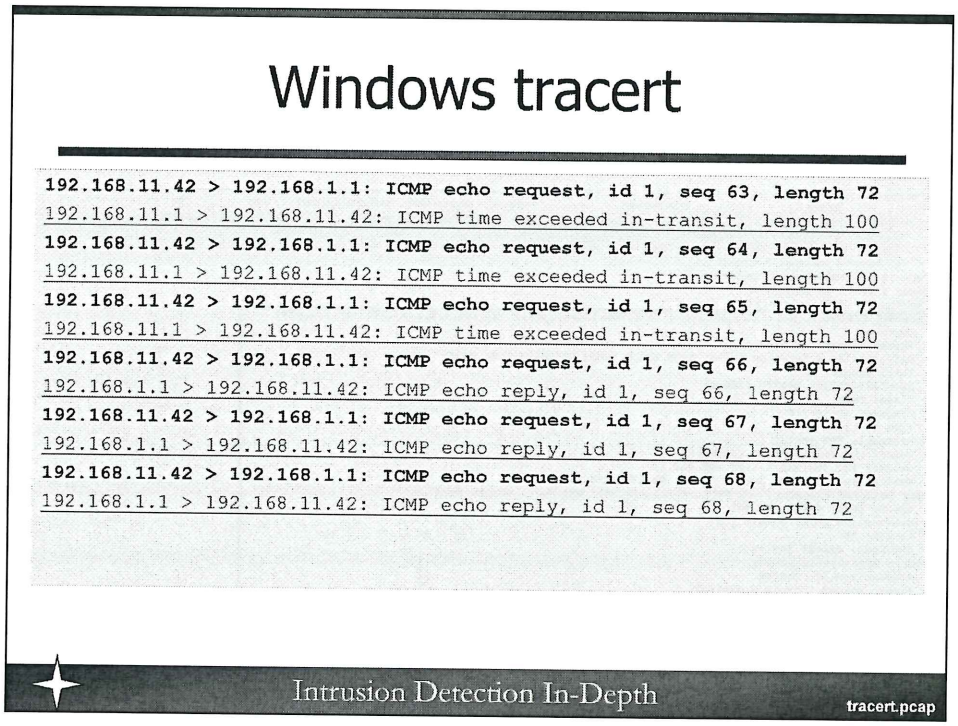


This is a UDP port unreachable error from 192.168.122.129 indicating that it does not listen on port 53. We have only two records here so we know that this error is associated with the previous DNS query. However, a host may send dozens, hundreds, or thousands of packets, necessitating a means of associating an ICMP error message with the packet that caused it.

The ICMP error message returned to the sender is supposed to have the sender's IP header and 8 bytes of the original datagram that caused the problem. Looking at the Wireshark output, it seems that you have a packet in a packet. The highlighted portion of the message represents the DNS query that the recipient host of the ICMP error can pair with the packet that it allegedly sent that caused the issue in the first place.

To display the output, enter in the command line:

wireshark dns-portunreachable.pcap



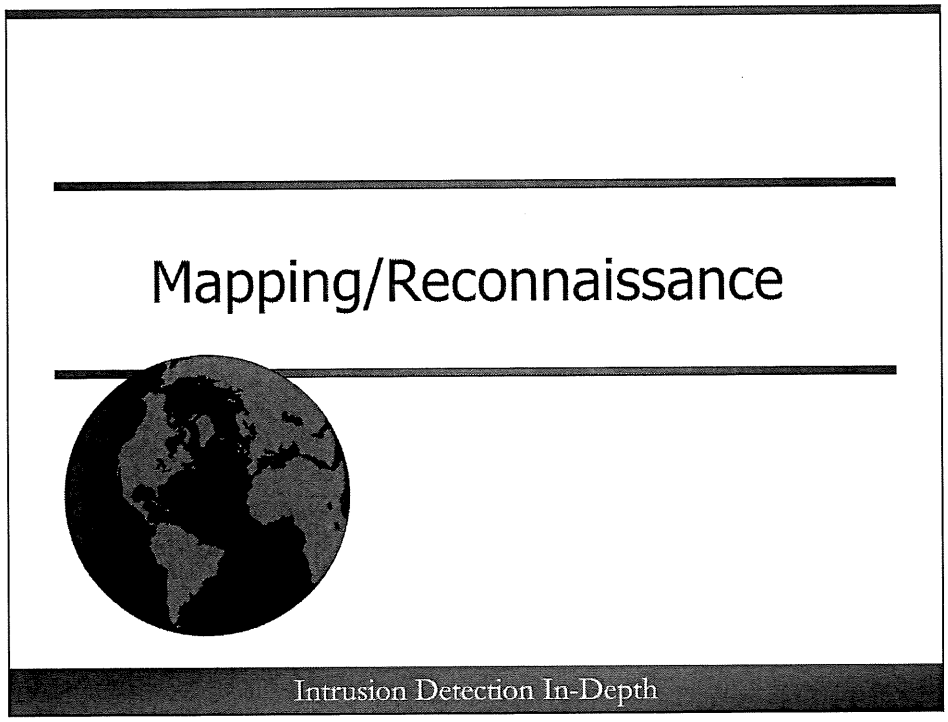
Windows tracert uses ICMP instead of UDP like the Linux traceroute to discover the routers that a packet traverses on its path from source to destination host.

Like Linux, it sends a series of three packets, in this case, ICMP echo requests with initial TTL values of 1. The first router returns an ICMP "time exceeded in-transit" message. The TTL value increments until the echo request eventually reaches the destination host that responds with an ICMP echo reply.

In the above example Windows host 192.168.11.42 wants to find all hops to 192.168.1.1. The first series of ICMP echo requests is returned with ICMP "time exceeded in-transit" messages. The TTL is incremented to 2 and the destination IP address of 192.168.1.1 returns ICMP echo replies.

To display the output, enter in the command line:

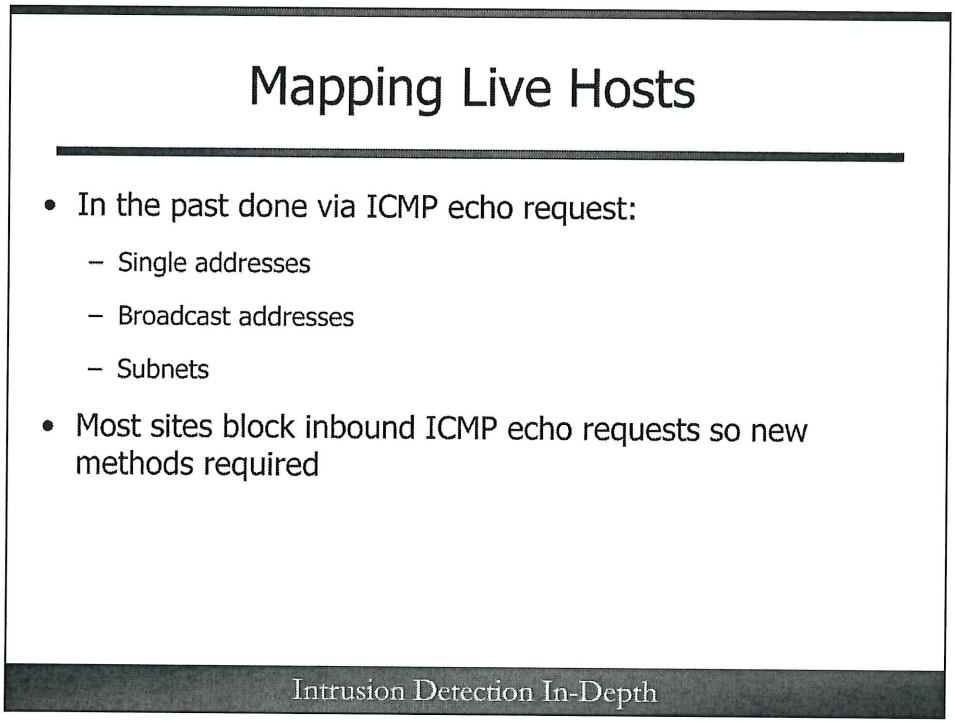
tcpdump -r tracert.pcap -nt



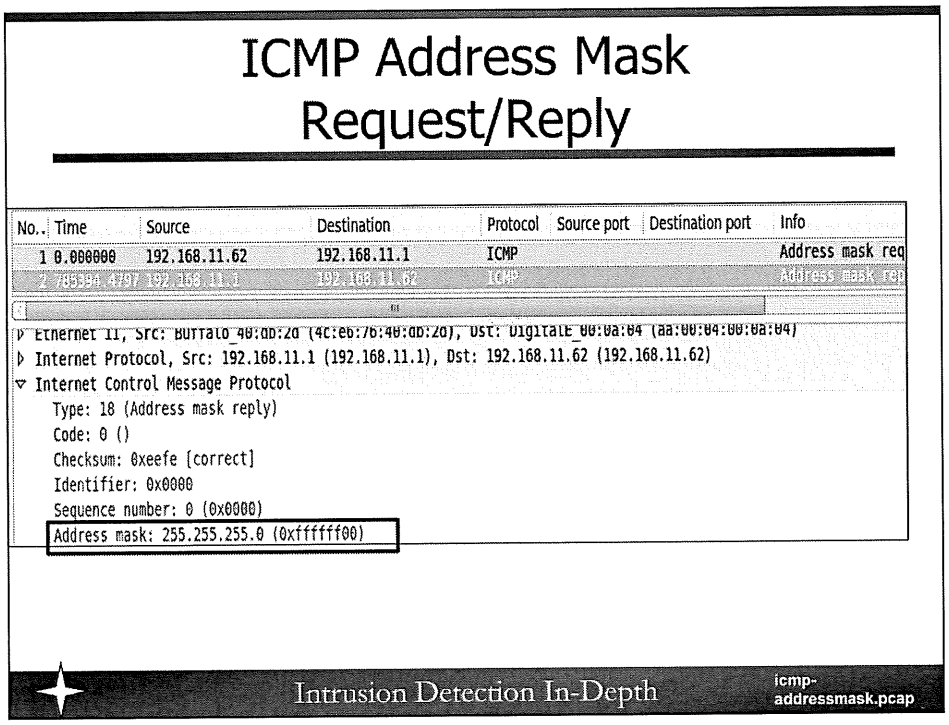
In this section, we'll show some of the mapping techniques using ICMP. Mapping is an important part of a planned attack since reconnaissance is typically the first step of most attacks. Mapping attempts to discover the IP numbers of live hosts in a network. Once found, any attack can be directed at the live hosts only.

You should know that since sending ICMP echo requests is one of the most common mapping techniques, many networks will block incoming ICMP echo requests. This has motivated the persistent attacker world to invent other scanning methods using other techniques or protocols.

Mapping is more practical in an IPv4 network because of the relatively small number of IPv4 addresses in many networks. The IPv6 128-bit address makes mapping and scanning impractical because of the large range of possible IPv6 addresses.



Mapping live hosts using an ICMP echo request to either a single, broadcast, or subnet address is likely to be blocked at most sites – either inbound or outbound. This was an effective mapping method, more so in the past. Today hackers have to get more clever about mapping techniques to successfully find live hosts.

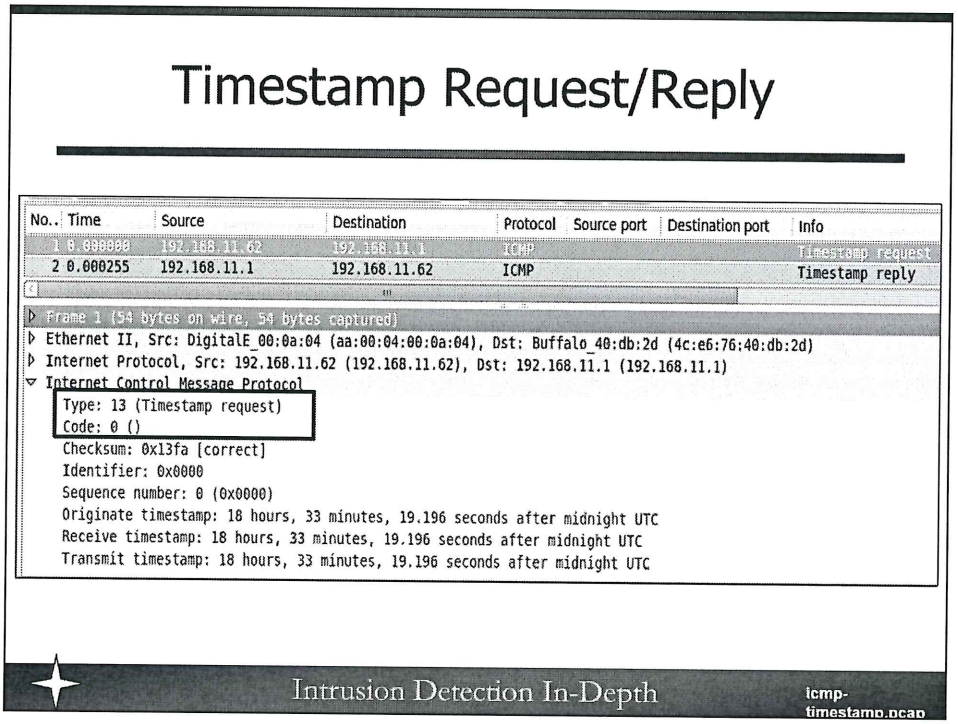


Suppose a network blocks ICMP echo requests, but fails to block other inbound ICMP traffic. For instance, an ICMPv4 address mask request (type 17) may not be blocked. The address mask queries for the subnet mask for the network on which the target host resides. This can provide insight into the configuration of a network and certainly identify a responding router.

192.168.11.62 sends and address mask request to 192.168.11.1. The 192.168.1.1 router responds with an address mask of 0xff ff ff 00, indicating that this network is a 192.168.1/24 network. The address mask request may elicit responses from gateway hosts and other network devices only. You may see such activity as a precursor to more scanning.

To display the output, enter in the command line:

wireshark icmp-addressmask.pcap

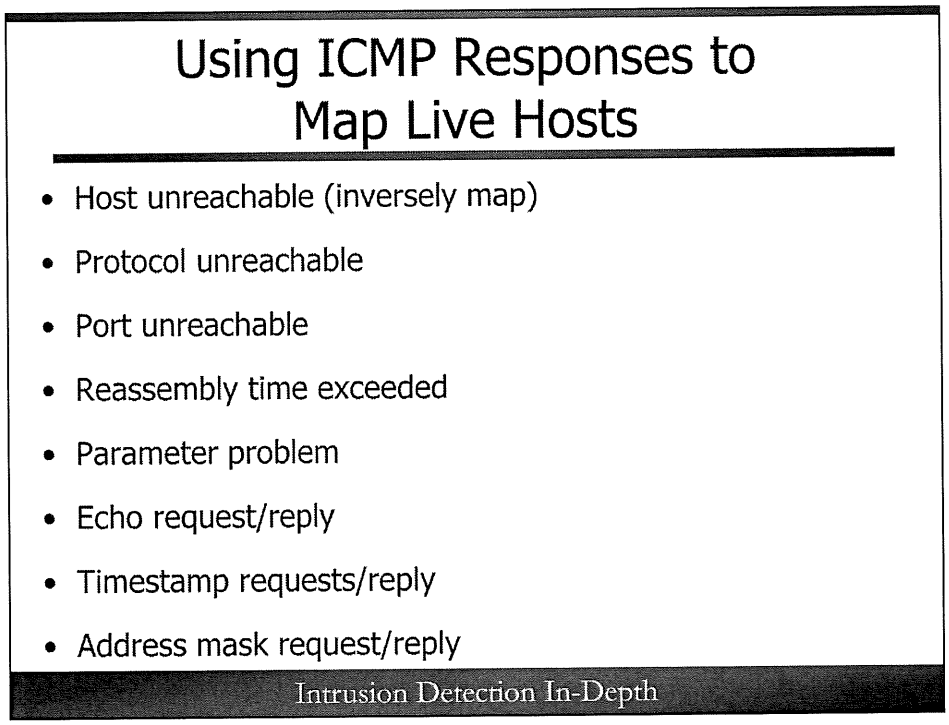


Another ICMP exchange involves a timestamp. This is different from the TCP timestamp and different than the timestamp that you see when you display output using tcpdump. The TCP timestamp is used to measure round-trip times for packets. Unless the hosts had some type of time synchronization applied, such as by network time protocol (NTP), it could not be used to accurately compute the individual time taken from sender to receiver and then compute the time back from receiver to sender.

The sending ICMP time request includes the sender's timestamp, and it is returned with the time that the destination host received the request, and a third field contains the time that the destination host returned the request. Using these three fields, hosts can compute the round-trip time. The timestamps displayed above are from a local network so they are identical.

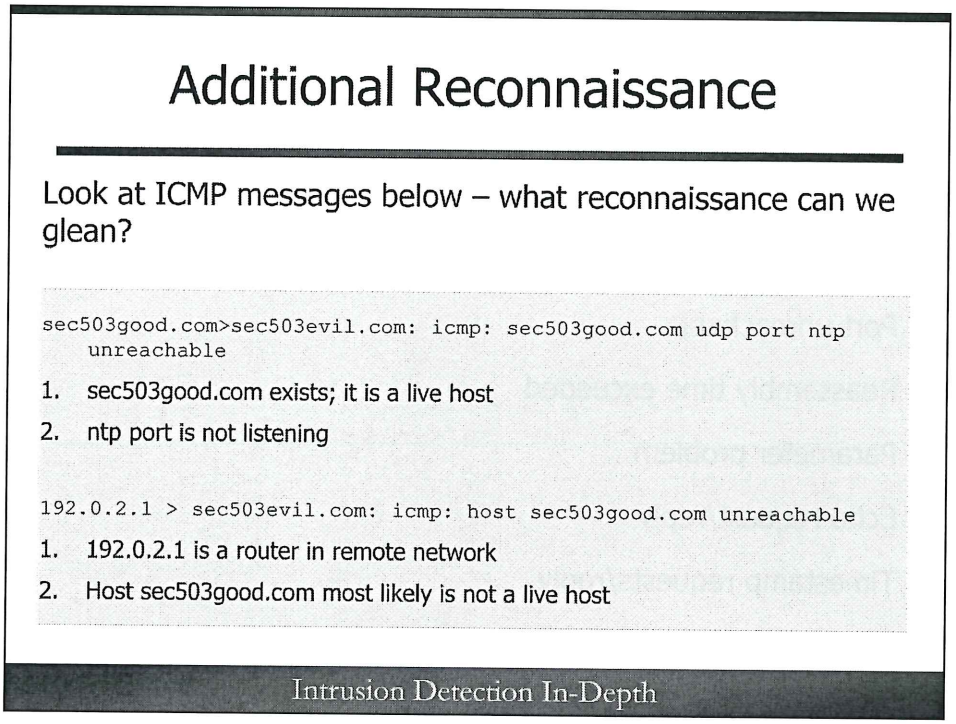
What kinds of reconnaissance can an attacker obtain from timestamps, assuming that the timestamps are somewhat accurate? First, she/he may have an idea of your time zone or where you may be geographically located. Second, there may be the possibility of identifying better times to attack when there is less scrutiny – perhaps weekends, nights, and holidays. Finally, examining many timestamps on a given network may give the attacker some insight into whether or not Network Time Protocol (NTP) is being used. If not, there may be some issues when trying to perform log correlation.

To display the output, enter in the command line:
wireshark icmp-timestamp.pcap



If any of the above ICMP messages is elicited in some way, hosts in a given network can be mapped to see if they are alive. If all you receive are "host unreachable" messages, the IP address is either not used, in which case you can conjecture that if you don't receive this message, the host is alive, or the host is temporarily down or incapable of responding. A "protocol unreachable message" will alert you to a live host as will "port unreachable", "reassembly time exceeded", or "parameter problem" messages. The more direct methods of sending ICMP requests whether echo, timestamp or address mask will also alert you that a responding host is alive.

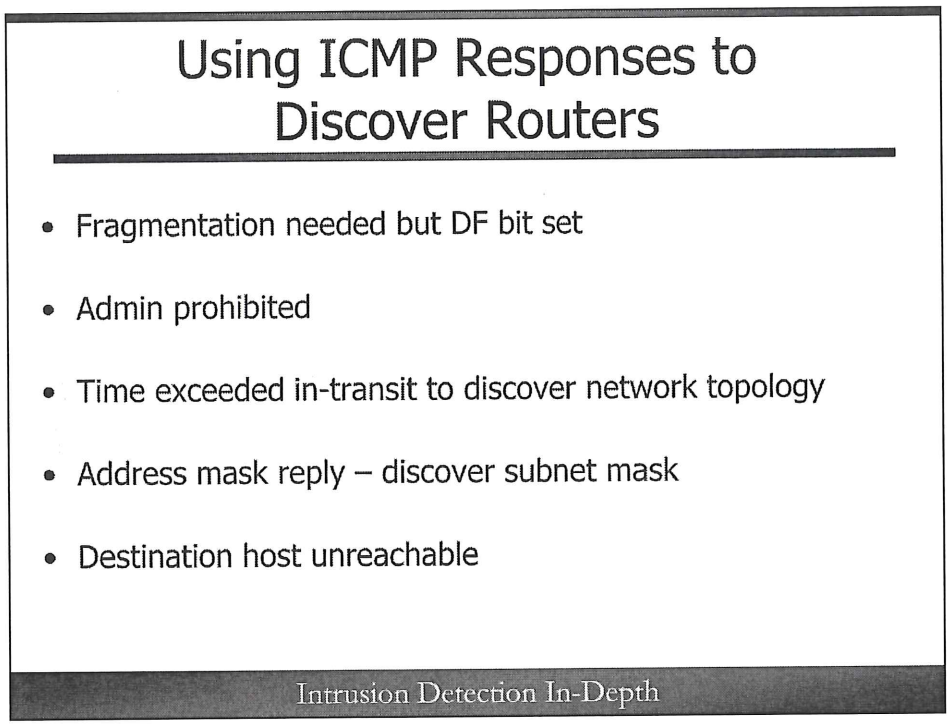
When we discussed UDP port scanning with Nmap, we saw some of the problems encountered with trying to map using ICMP messages as positive confirmation of a port being closed. The same can be said of trying to map networks using ICMP error messages as confirmation of a host's existence and no ICMP error message received would imply that the host does not exist. Again, you don't know if packets have been dropped on the way to the target host or blocked on entry or egress to the network. It is very difficult to determine if a lack of response means that the host does not exist or it just means that you were unable to elicit or receive a response.



Many times, we've mentioned that ICMP messages contain a lot of information about a network or host if someone wants to try to gain additional reconnaissance. Looking at the top two records, the first ICMP message informs us that a UDP port on the target host is not listening. By virtue of the fact that the host responded, it means that the host is a live host. So, we get two pieces of information about the remote host.

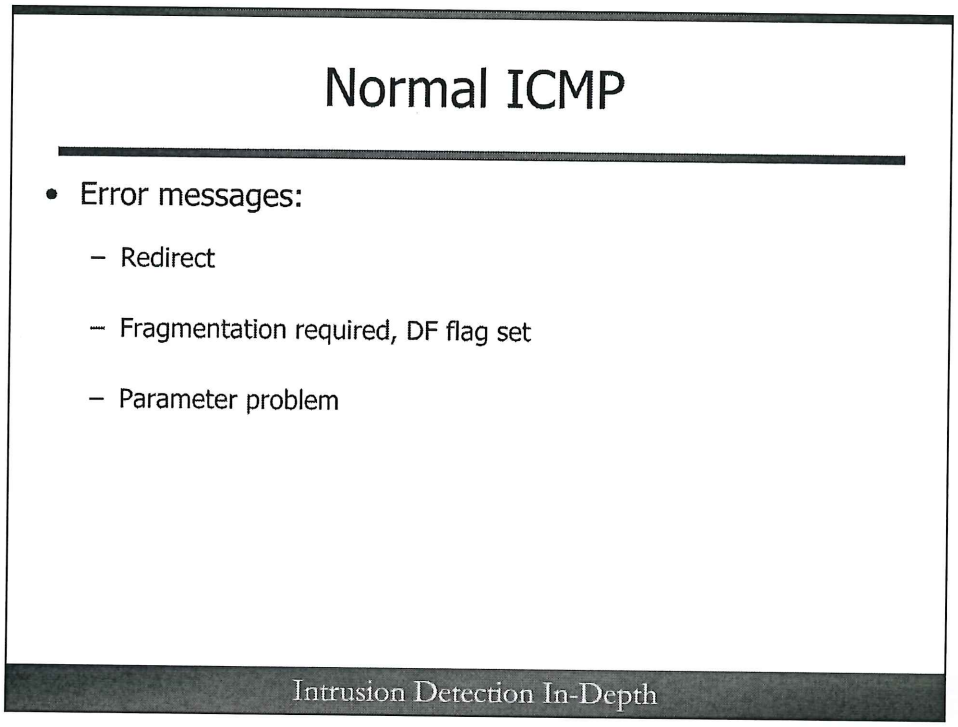
The second message that informs us that host sec503good.com is unreachable and potentially contains more than meets the eye, as well. If we see a message like this, we know that the sender is a router. This is a helpful piece of information to have if a scanner is trying to discover the topology of your network. Also, we can be pretty certain that host sec503good.com is not a live host. Sure, it's possible that it is temporarily out of commission, but more likely, it is not a host that a smart scanner would attempt to attack.

There are plenty more ICMP messages that contain a wealth of information if only they can be elicited. Scanning UDP ports is done via attempting to elicit an ICMP "port unreachable" message. Live UDP ports do not give any response unless we actually look at the application level. We can also discover the protocols that a particular host offers by trying to elicit "protocol unreachable" messages. So, you might see where we can use this "inverse" logic to map traits of a host or network. We assume (sometimes incorrectly so) that the absence of an ICMP message means the affirmative (a listening UDP port or an offered protocol), and the presence of an ICMP message means the negative.

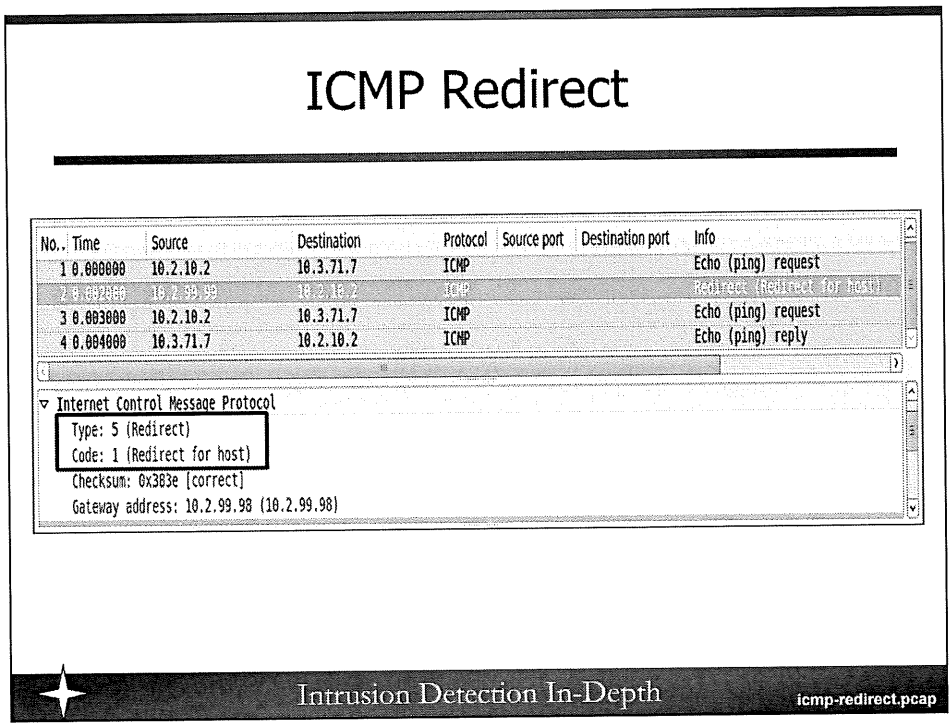


Some of these ICMP error messages are exclusively issued by routers. The address mask can be from a router or host acting as a router. The "fragmentation required but DF bit set" will allow discovery of the MTU of a link in the network. Most routers will reply with the MTU that caused the problem. The "admin prohibited" messages inform that a packet was not allowed into the network. If you examined the embedded message after the ICMP header, you could discover the blocked protocol/port.

The "time exceeded in-transit" message used by traceroute/tracert can help with router discovery. By incrementing the initial TTL by 1 and repeating the same connection, discovery of the hierarchy of routers in a network and which hosts they control may be possible. The address mask reply will allow discovery of the subnet mask of the router that responds. A "destination host unreachable" message is typically sent by the router too.



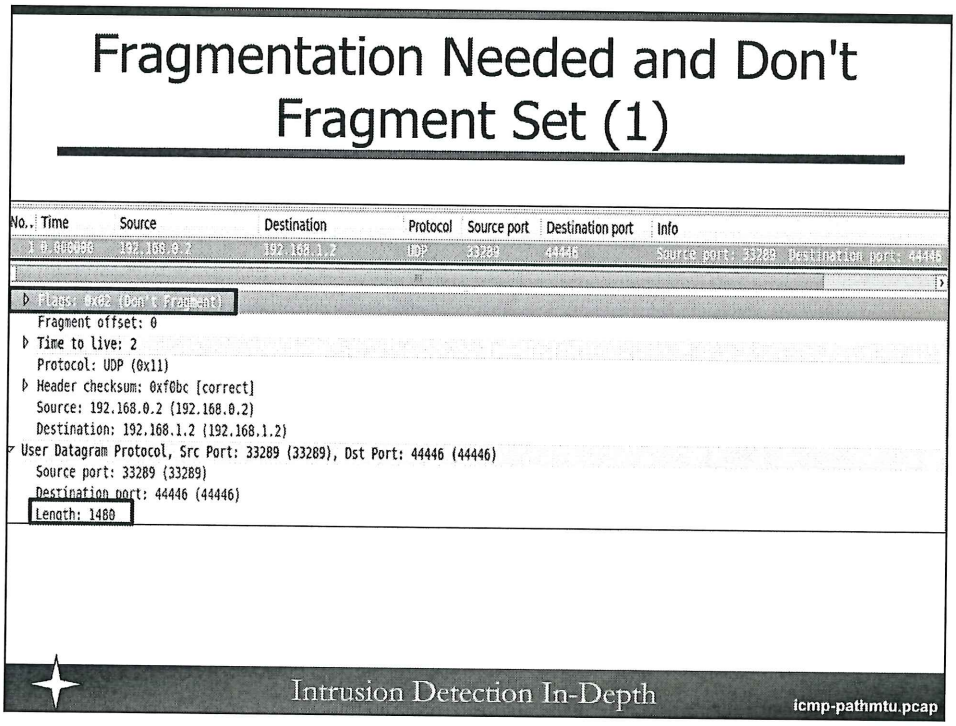
We've covered many common ICMP messages in previous material. We'll look at some of those messages that we haven't covered.



Host 10.2.10.2 would like to ping host 10.3.71.7. On its way to the destination, it travels through router 10.2.99.99. This router believes that a better router to use next time would be 10.2.99.98. Router 10.2.99.99 forwards the echo request to the destination, but informs host 10.2.10.2 to make a change in its routing table to use the gateway 10.2.99.98 when sending subsequent packets to 10.3.71.7.

To display the output, enter in the command line:

wireshark icmp-redirect.pcap

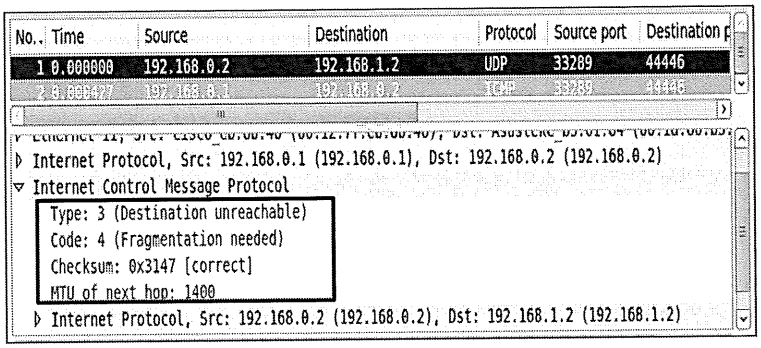


What happens if a packet must be fragmented, yet the Don't Fragment flag is set? This is a problem for IPv4 intermediate routers. For instance, host 192.168.0.2 wants to send an oversized packet to 192.168.1.2 yet has the DF set. It has 1480 bytes of UDP header and data to send.

To display the output, enter in the command line:
wireshark icmp-pathmtu.pcap

This is the first record.

Fragmentation Needed and Don't Fragment Set (2)

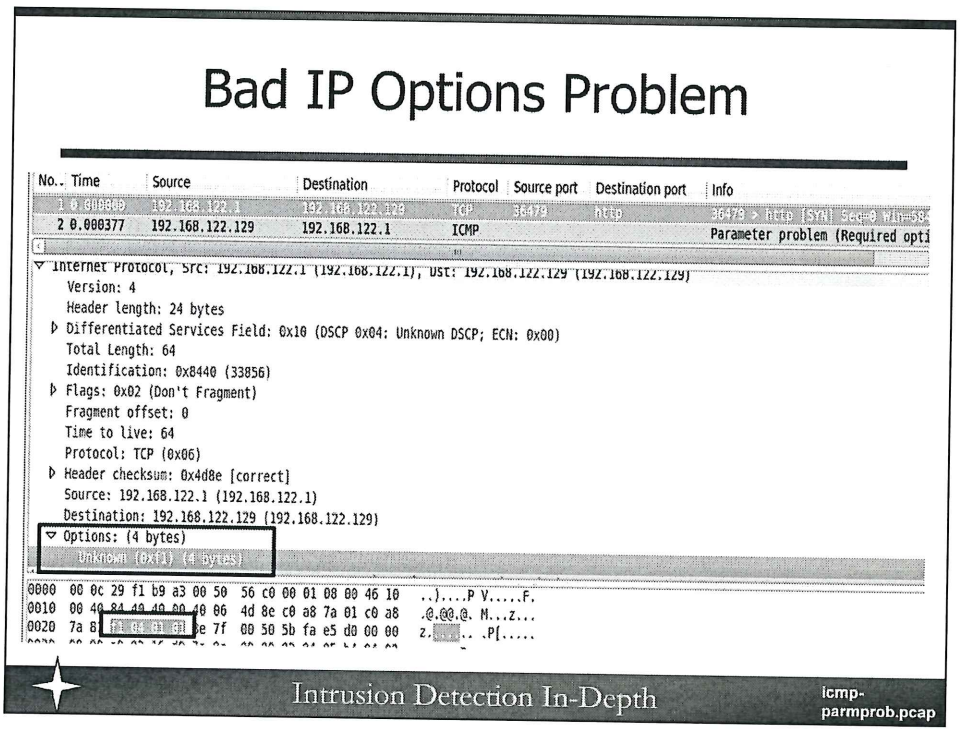


There is a 1400 byte MTU for a network en route to 192.168.1.12. The router 192.168.0.1 informs the sender that "fragmentation is needed" for the smaller MTU of 1400.

To display the output, enter in the command line:

wireshark icmp-pathmtu.pcap

This is the second record.

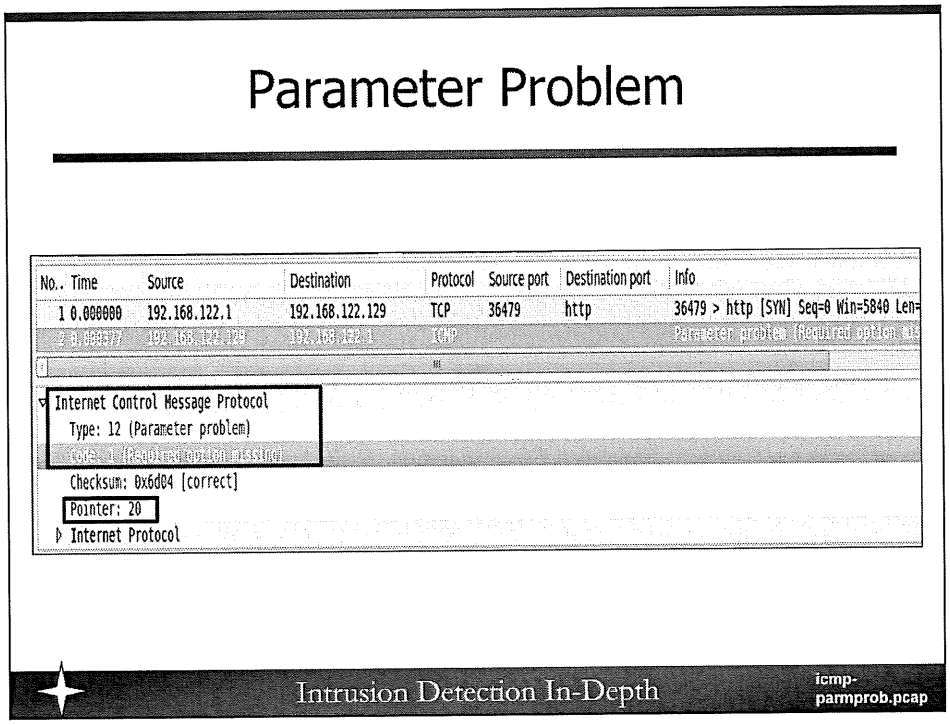


There is a catchall ICMP error message known as the "parameter problem". Vague guidance is given for when it is to be used, however, it is supposed to be used if an unknown IP option is discovered. The ICMP "parameter problem" message includes a pointer to the beginning of the problem byte – counting begins at 0.

In the first record above, host 192.168.122.1 attempts to send a SYN packet to 192.168.122.129 with an invalid IP option of 0xf1.

To display the output, enter in the command line:

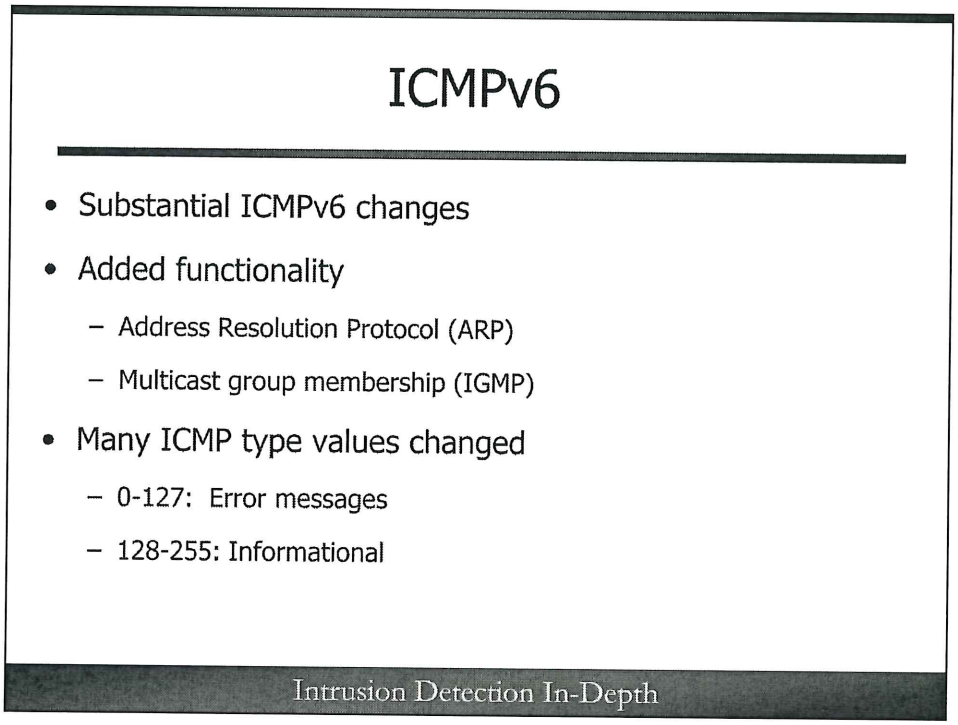
wireshark icmp-paramprob.pcap



Host 192.168.122.129 cannot accept the packet because of the invalid IP option. It returns an ICMP "parameter problem" error and points to byte 20 as the issue.

To display the output, enter in the command line:

wireshark icmp-paramprob.pcap

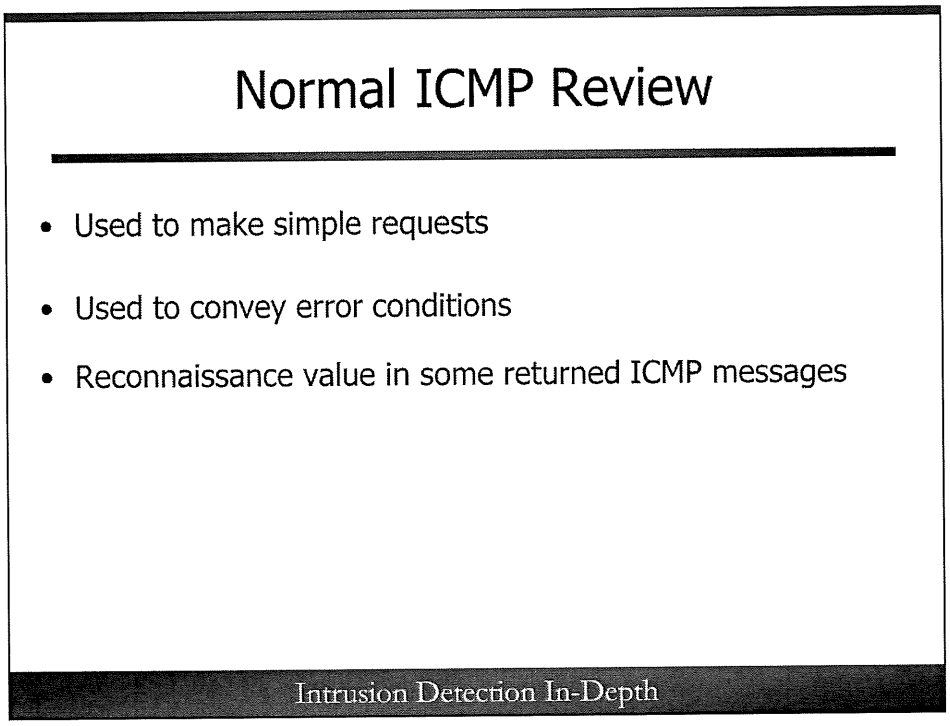


TCP and UDP are no different under IPv6 as they are under IPv4. In other words, no new or altered functionality is found in these protocols (except for jumbogram specification) as well as most other protocols that ride over IP. However, ICMPv6 has changed significantly as we've learned. Let's just review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

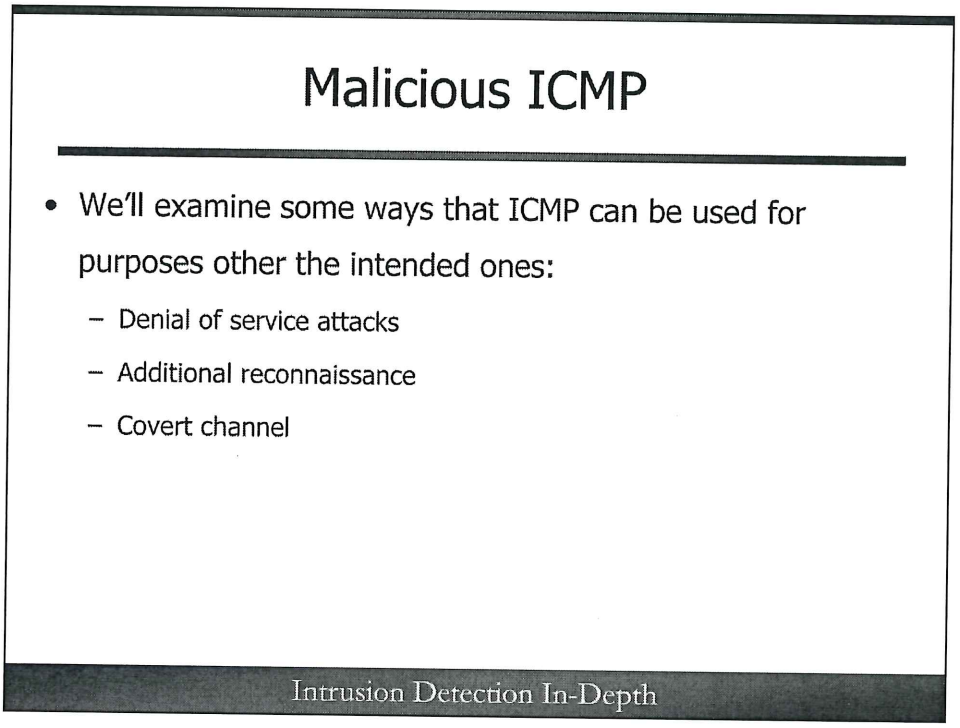
First, Neighbor Discovery Protocol (NDP) provides hosts on a local network a function similar to ARP in finding an address of a host or node on the same link. But unlike ARP, it also provides a means to determine what nodes are reachable and it also has the capability to detect changes in link-layer 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 management (IGMP) was assigned its own protocol value of 2. In IPv6, multicast functions are incorporated into ICMP known as Multicast Listener Discovery (MLD). 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.

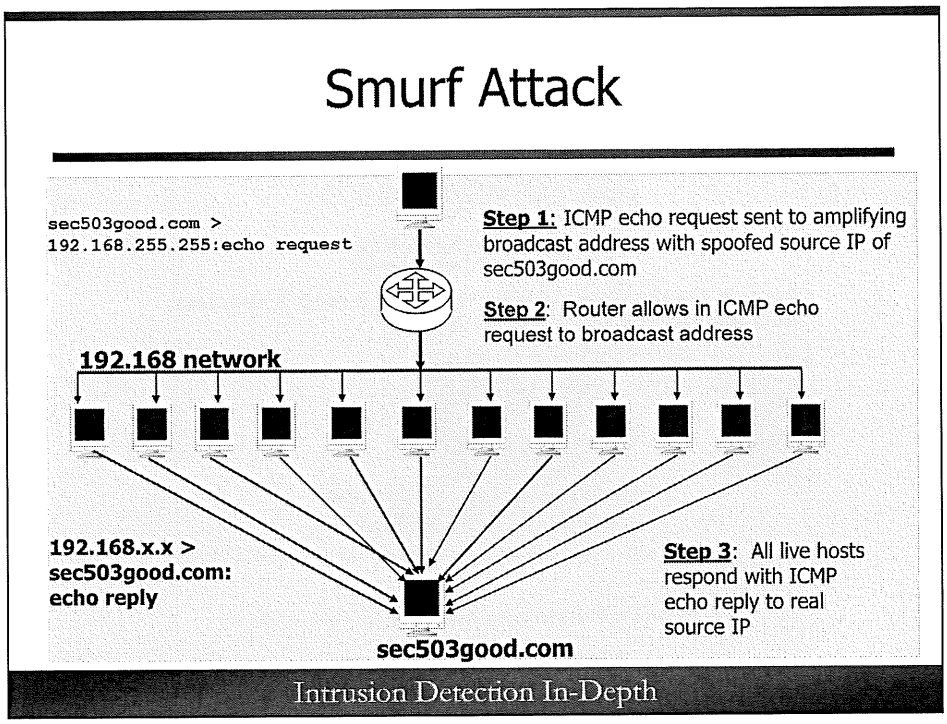
You'll also find that the standard IPv4 ICMP types have changed. The ICMPv4 type 8 is an echo request and type 0 is an echo reply. Now, ICMPv6 type 128 is an echo request and type 129 is an echo reply. Error messages have type values less than 128 and should not be blocked. Informational messages are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



We examined many of the ICMP messages that you may see while monitoring your network. We saw many of the different informative ICMP error messages. As you noticed, these can be sent both by hosts or routers that discover a problem. We also discussed the notion that some of the ICMP unreachable errors are best disabled if you are concerned about the reconnaissance information that may be gathered from them.



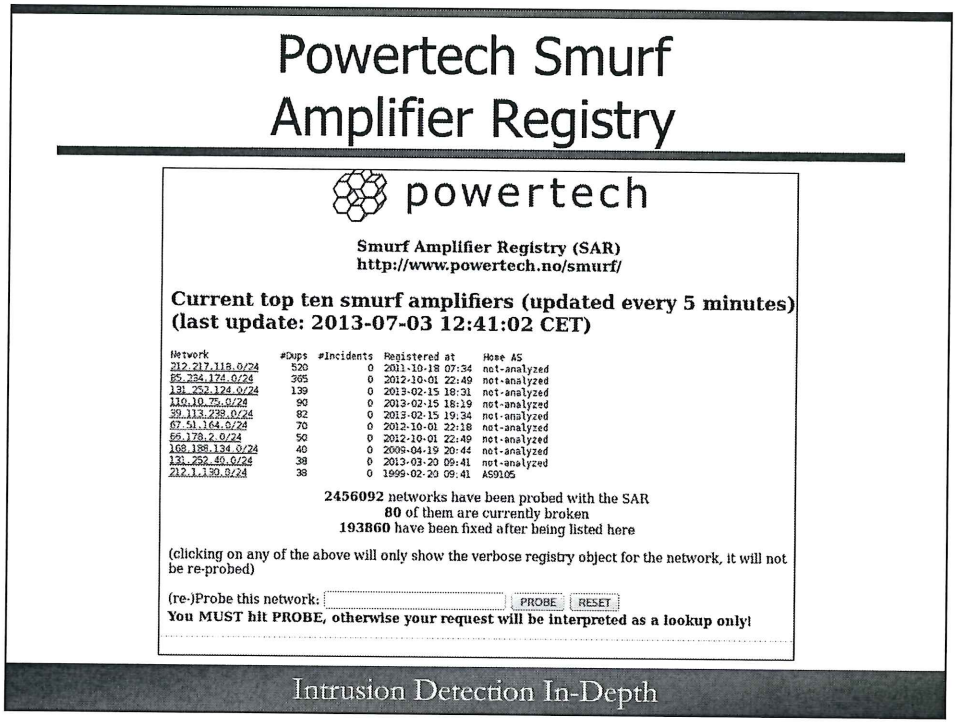
Just like many of the other protocols, ICMP can be used for malicious or perhaps more accurately in some cases – not the intended purpose. This section discusses some of the ways that ICMP has been used for reconnaissance, as a conduit for denial of service attempts, and as a tunneling protocol.



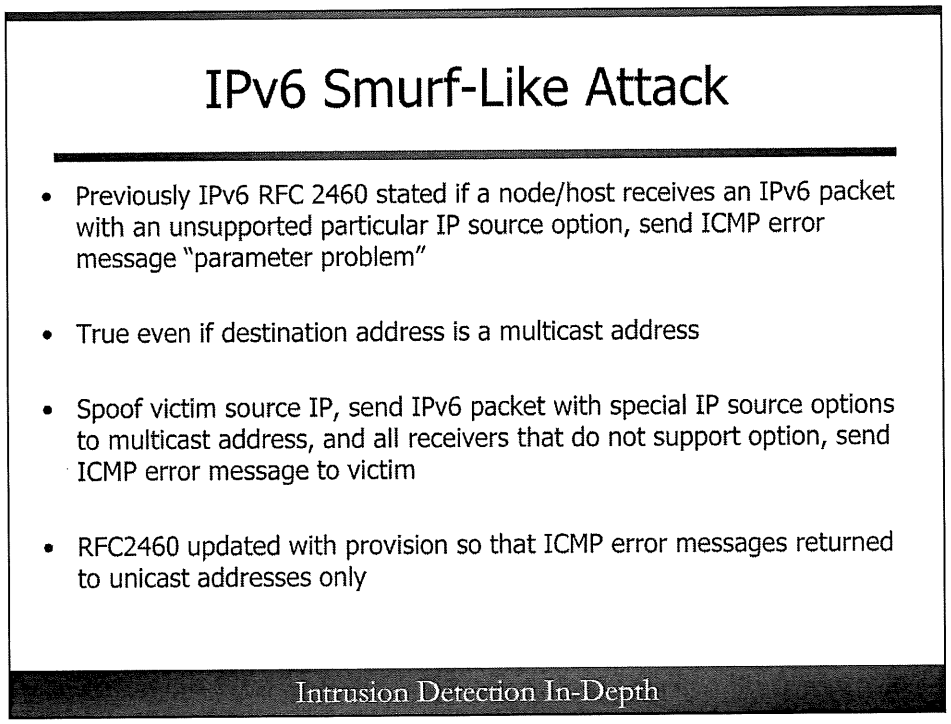
The infamous Smurf attack is displayed in this slide. Remember that ICMP can be broadcast; many hosts can listen and respond to a single ICMP echo request sent to a broadcast address. This capability is used to execute a denial of service against a hapless target host or network.

First, a malicious host crafts an ICMP echo request to a broadcast address of an intermediate network with a spoofed source IP of the target host. Next, the intermediate site has to accept the network broadcast activity. If it does, the ICMP echo request is sent to all hosts on the given broadcast subnet. Finally, all of the live hosts in the intermediate network that respond will send an ICMP echo reply to what they believe to be the sender, or the target host. Now, if the malicious user sends many of these, if the intermediate site is large and has many responding hosts, and if the target site is serviced by a smaller pipe, the target host or network on which it resides can become choked with all the activity and suffer a degradation or denial of service.

So, here is another reason that you may want to deny broadcast traffic from entering into your network, precluding your site from being used as a Smurf amplification network.



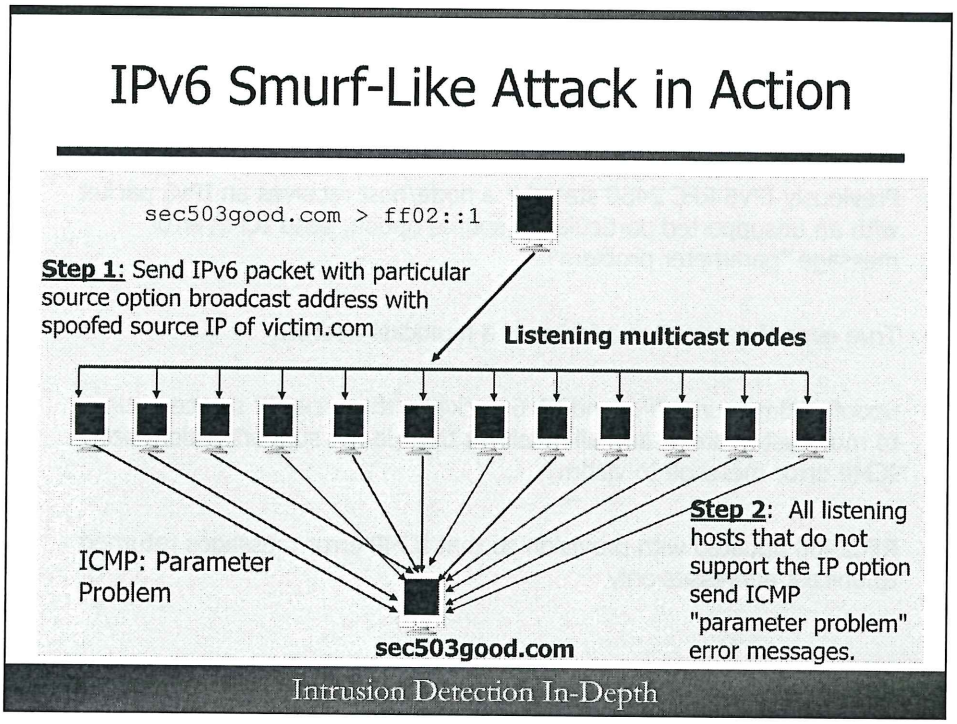
The site http://smurf.powertech.no offers a list of the top ten current Smurf amplifiers. It also offers you the capability to enter an IP address to check to see if your network can be used as a Smurf amplifier. As you can imagine, you are probably not the only one trying to discover whether or not your network can be used as a Smurf amplifier. Even though the Smurf attack first surfaced years ago, as you can plainly see, there are still networks that allow inbound broadcast traffic.



An IPv6 packet may have IP options in the extension headers Hop-by-Hop Options or Destination Options. RFC 2460, that gives guidance for IPv6, states that if certain IP options found in these particular extension headers are not supported by a receiving node or host, an ICMP error message "parameter problem" is to be returned to the sender. This is applicable if the one byte IP option field in the extension header has a value where the two high order bits are "10". In other words, the individual bit values in the IP Option byte would be "10xx xxxx" where "x" is 0 or 1.

In IPv6 a multicast address is one sent to some or all, depending on the exact multicast address, the hosts in broadcast range. Suppose an attacker spoofs an IPv6 packet from the victim host to all listening multicast hosts that causes the ICMP error to be returned to the victim. The returned aggregate ICMP error messages could potentially cause a DoS of the victim host.

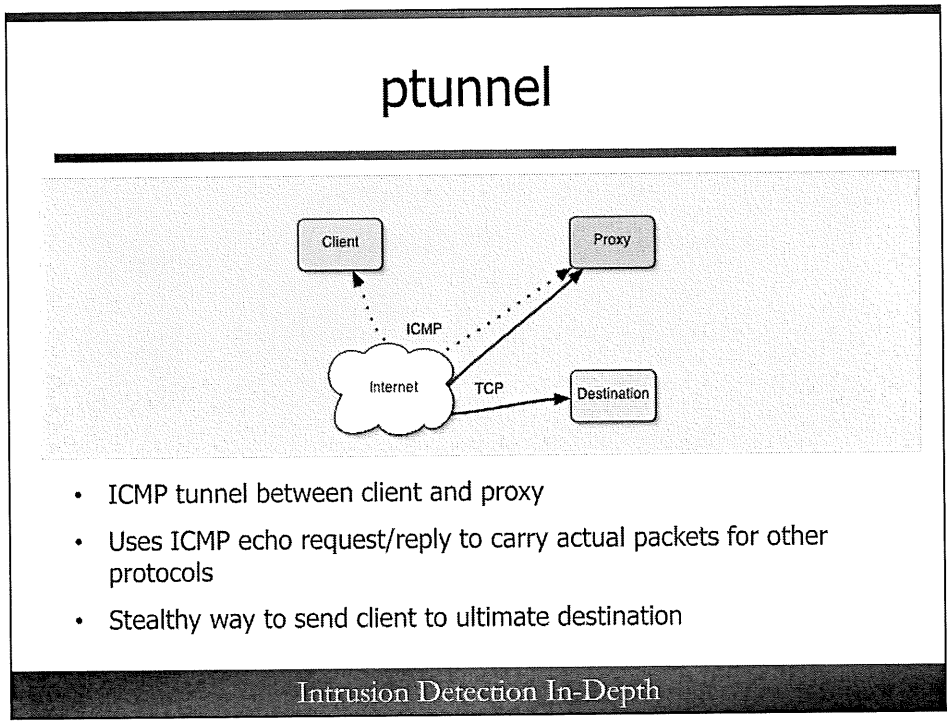
RFC2460 has since been updated to indicate that the ICMP "parameter problem" error message is to be sent only if the destination address is a unicast address.



This is a depiction of the IPv6 Smurf-like attack. It starts with an attacker who spoofs an IPv6 packet with an IP option value in the extension header with the two high-order bits of '10'. The source IP in the packet is the victim host and the destination is the multicast address of ff02::1 (all hosts) within the network.

Any listening host in the multicast range attempts to process the packet and its extension header IP option. It sends an ICMP "parameter problem" error to the victim host if it does not support that option. Given enough listening hosts and perhaps synchronized attacks, it may cause a DoS for the victim host.

review

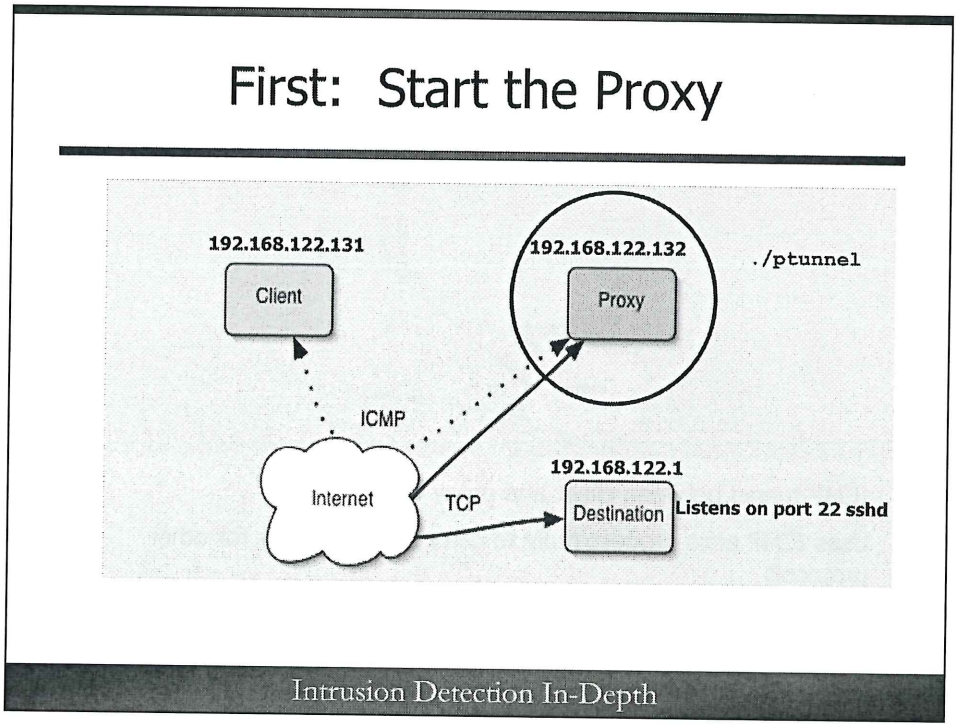


A current ICMP tunneling or covert channel package is known as ptunnel. Unlike an older ICMP tunneling program named Loki, that directly sends traffic between two hosts, ptunnel uses a proxy to forward traffic. The proxy acts more like a host that interprets the type of connection/protocol that the client wishes to use to connect to an actual destination host. This allows the client to "hide" the type and content of the connection to the actual destination in an ICMP tunnel.

One of the mentioned uses was to circumvent pay-per-use WiFi in public places. Typically, when you attempt to connect to an open hotspot, you'll receive an IP address via DHCP. When you attempt some kind of TCP connection like HTTP traffic via your web browser, you'll be sent a page that asks for a credit card number. Unless you pay, you will not be able to send or receive TCP traffic. However, many of these sites allow ICMP traffic unimpeded even without paying. The user then employs the ptunnel client portion of the software to establish an ICMP tunnel to the ptunnel proxy that (s)he controls. The proxy is responsible for taking the ptunnel payload and turning it into a TCP connection to the destination host.

The traffic between the client and proxy is all ICMP. The traffic between the proxy and the destination is the actual session. If you want to read more about ptunnel look at:

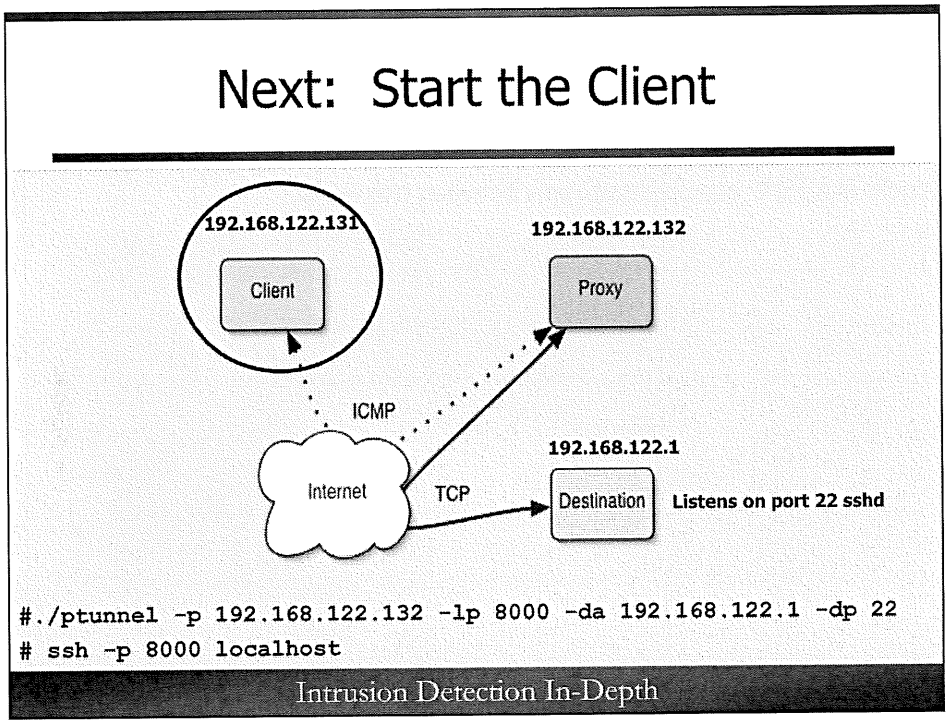
http://freshmeat.net/projects/ptunnel/



Client is 192.168.122.131
Proxy is 192.168.122.132
Server is 192.168.122.1

A ptunnel session is started on host Proxy as follows:

./ptunnel
[inf]: Starting ptunnel v 0.72.
[inf]: (c) 2004-2011 Daniel Stoedle, <daniels@cs.uit.no>
[inf]: Security features by Sebastien Raveau, <sebastien.raveau@epita.fr>
[inf]: Forwarding incoming ping packets over TCP.
[inf]: Ping proxy is listening in privileged mode.
[inf]: Incoming tunnel request from 192.168.122.131.
[inf]: Starting new session to 192.168.122.1:22 with ID 61093

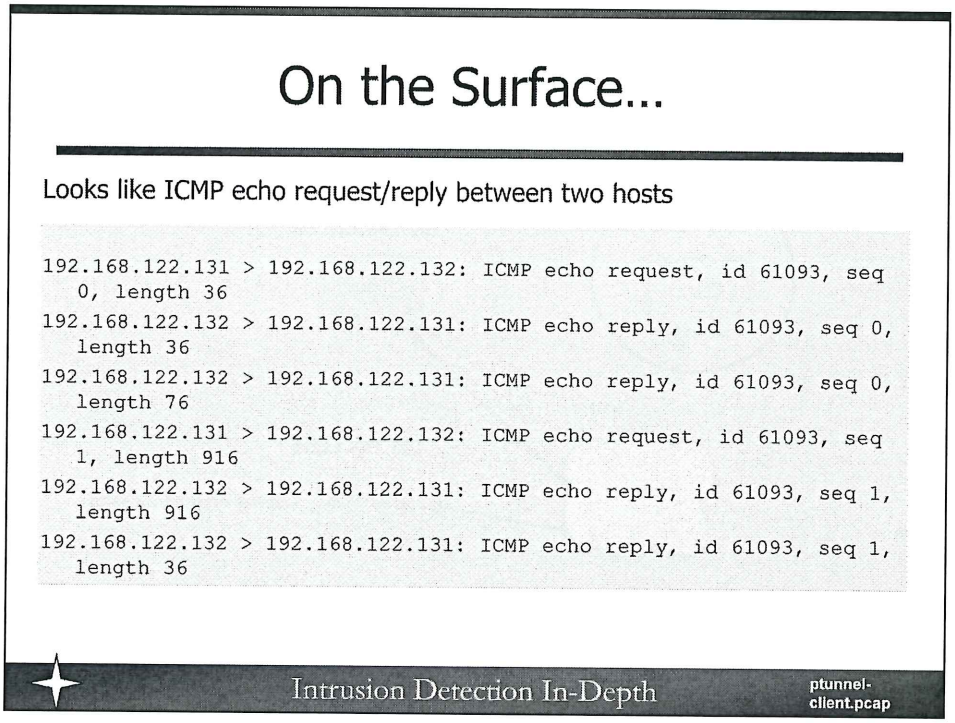


Here are the commands and output from running ptunnel on the client:

./ptunnel -p 192.168.122.132 -lp 8000 -da 192.168.122.1 -dp 22

[inf]: Starting ptunnel v 0.72.
[inf]: (c) 2004-2011 Daniel Stoedle, <daniels@cs.uit.no>
[inf]: Security features by Sebastien Raveau, <sebastien.raveau@epita.fr>
[inf]: Relaying packets from incoming TCP streams.
[inf]: Incoming connection.
[evt]: No running proxy thread - starting it.
[inf]: Ping proxy is listening in privileged mode.

ssh -p 8000 localhost
The authenticity of host '[localhost]:8000 ([127.0.0.1]:8000)' can't be established.
RSA key fingerprint is 9a:31:df:c3:ee:83:b5:eb:c4:3d:fd:58:71:f4:02:77.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '[localhost]:8000' (RSA) to the list of known hosts.

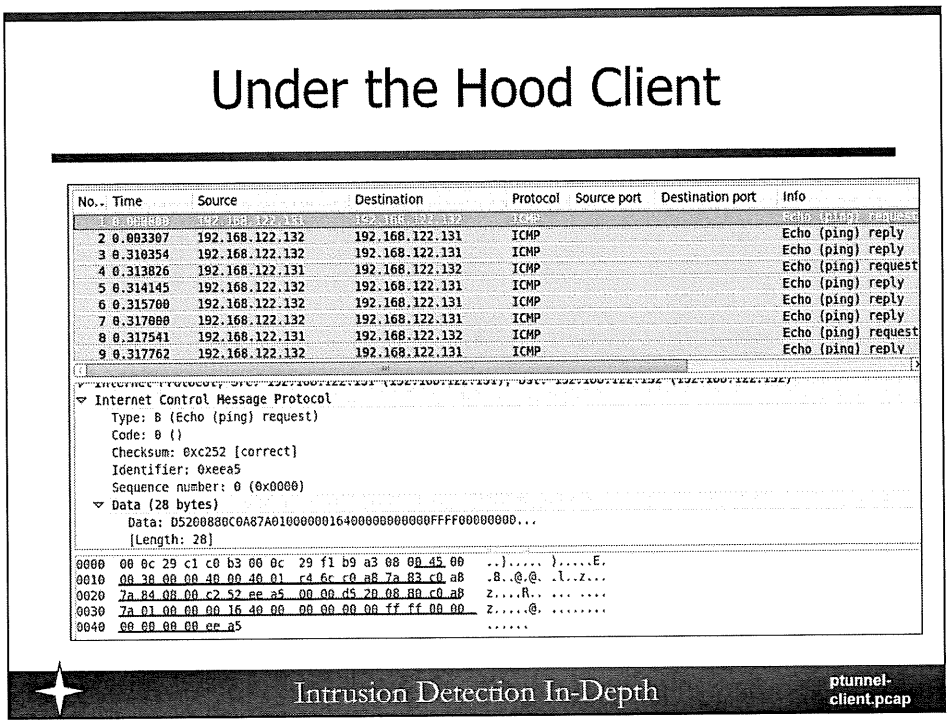


If you captured traffic on the client going to the proxy, you would see only ICMP echo requests and replies. But, there are a couple of oddities about this ICMP traffic. Can you see two things that might identify this as unique or perhaps tunnel traffic?

First, there is a mismatch between echo requests and echo replies. You see two or more replies for each of the displayed requests above. This may be indicative of a tunnel. Next, take a look at the sizes of the payload in the some of the traffic. For instance, 916 or 820 bytes are longer than normal ICMP messages. Finally, the payloads of an echo request/echo reply pair should be identical, therefore the same length.

To view the output, enter on the command line:

tcpdump -nt -r ptunnel-client.pcap -c 10

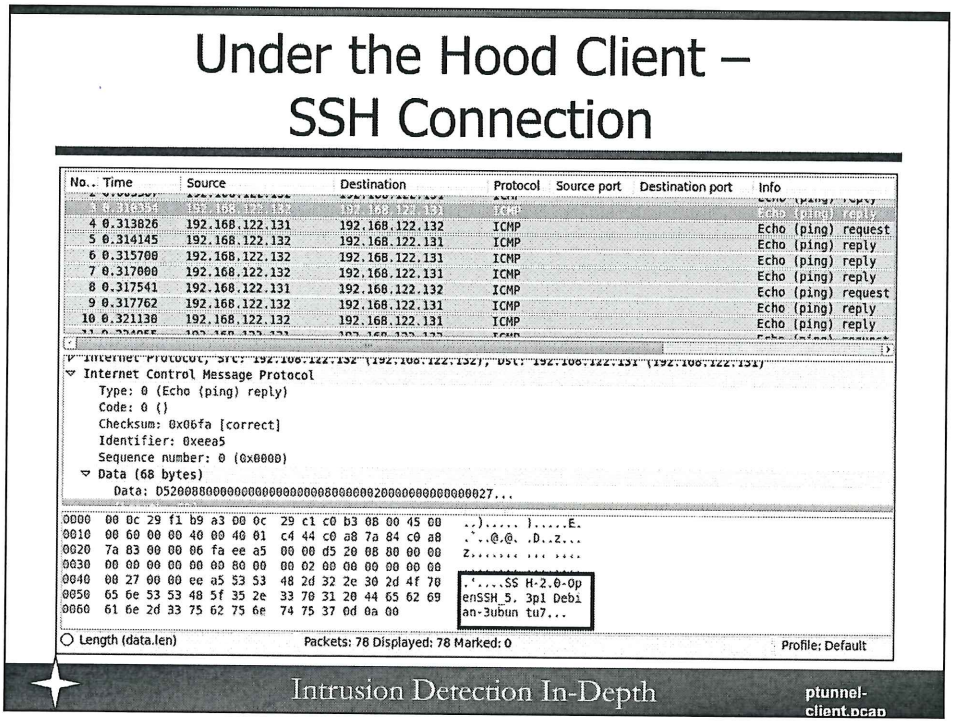


We see some of the telltale signs of ptunnel in the the data portion of this ICMP echo request.

The session is established using the ICMP ID of 61039 (0xeea5) as the ptunnel session identifier. You see this hex value both in the ICMP header of the packet and the payload portion of the ICMP message. You can see the tunneled packet in the underlined hex values in the bytes pane of Wireshark. There are also codes in the data to tell if the message is starting a new proxy session, contains data, acknowledges receipt of packet, closes the session, or is authentication related.

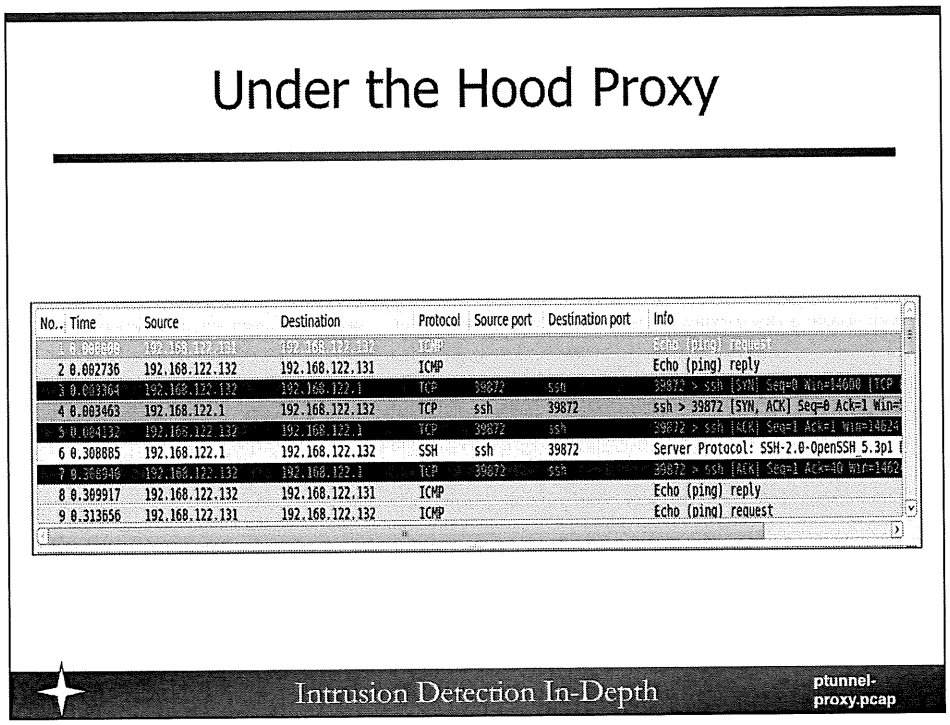
As you might imagine, ptunnel can be very slow since it needs to make two connections (client to proxy, proxy to server) rather than connect directly and requires encryption along the way if the server connection is SSH.

To view the output, enter on the command line:
wireshark ptunnel-client.pcap



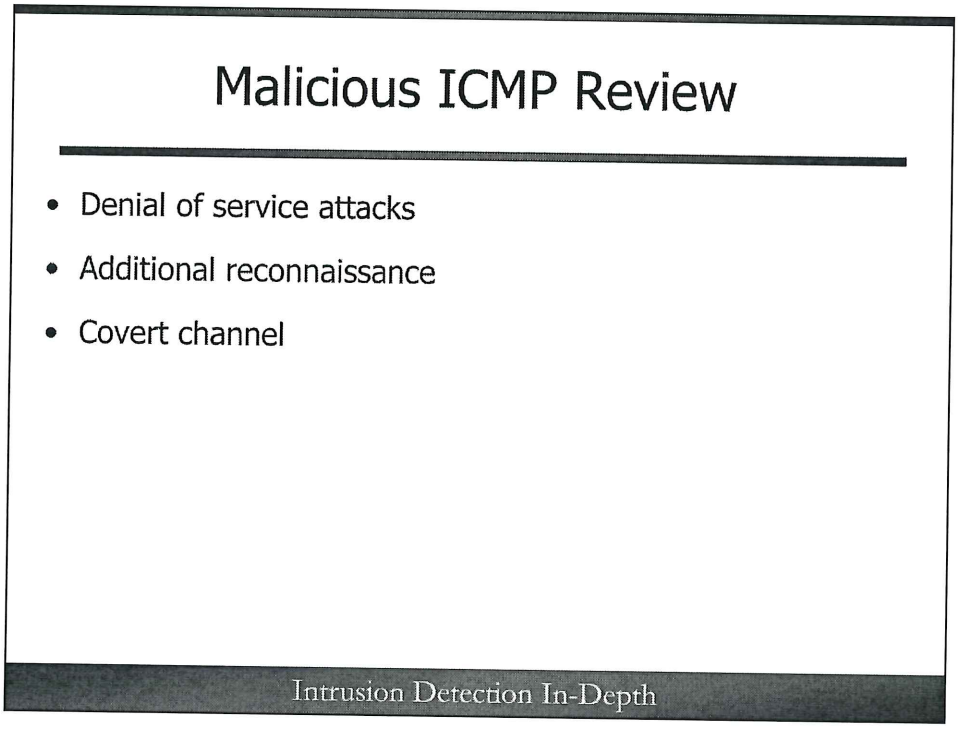
The third packet in the exchange – an ICMP echo reply shows a portion of the sshd connection. This has been passed from the proxy back to the client as an ICMP reply.

To view the output, enter on the command line:
wireshark ptunnel-client.pcap



If you look at the traffic from the proxy host, you see the entire exchange. The first two packets are echo request and replies to begin the tunnel between the client and proxy, then some SSH traffic from the proxy to the server, and finally the ICMP reply and request from the proxy to the client to deliver the SSH connection.

To view the output, enter on the command line:
wireshark ptunnel-proxy.pcap



Wrapping up this section, we learned that ICMP has been manipulated to be used for other purposes. We've seen where ICMP can be used for DoS attacks such as Smurf, and the IPv6 source option that permits a flood of ICMP parameter problem messages.

ICMP provides a lot of information about the network or hosts on the network and can be used for scanning purposes. Both routers and hosts offer information about different aspects of the state of listening hosts, ports, network topology and access control lists – to name a few.

We also discussed how ptunnel has completely altered the intended purpose of ICMP by using it as a tunneling mechanism for stealthy or malicious activity.

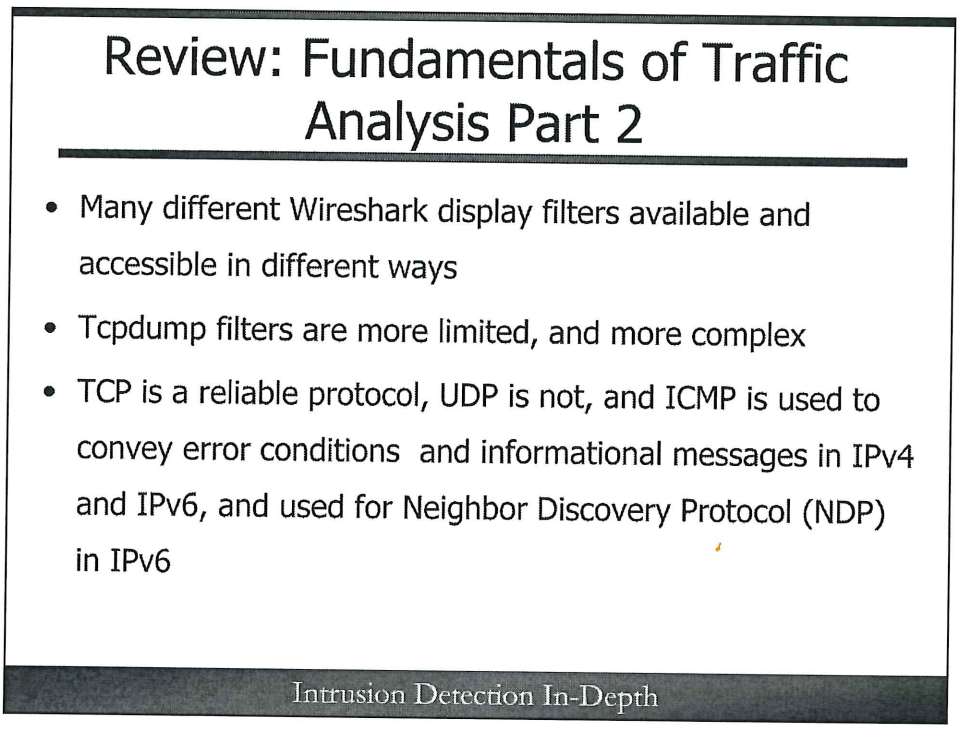
UDP-ICMP Exercises

Workbook

Exercise:	"UDP-ICMP"	
Introduction:		Page 54-B
Questions:	Approach #1 -	Page 55-B
	Approach #2 -	Page 58-B
	Extra Credit -	Page 60-B
Answers:		Page 61-B

Intrusion Detection In-Depth

This page intentionally left blank.



Let's wrap up with a summary of what was covered today. We began the day with learning to use Wireshark display filters to pick out records with some distinguishing characteristic. 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 them is straightforward; however, they become more 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 bit fields 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 a price – 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.

ABOUT SANS

SANS is the most trusted and by far the largest source for information security training and certification in the world. It also develops, maintains, and makes available at no cost the largest collection of research documents about various aspects of information security, and it operates the Internet's early warning system – the Internet Storm Center. The SANS (SysAdmin, Audit, Network, Security) Institute was established in 1989 as a cooperative research and education organization. Its programs now reach more than 165,000 security professionals around the world. A range of individuals from auditors and network administrators to chief information security officers are sharing the lessons they learn and are jointly finding solutions to the challenges they face. At the heart of SANS are the many security practitioners in varied global organizations from corporations to universities working together to help the entire information security community. SANS provides intensive, immersion training designed to help you and your staff master the practical steps necessary for defending systems and networks against the most dangerous threats – the ones being actively exploited. This training is full of important and immediately useful techniques that you can put to work as soon as you return to your office. Courses were developed through a consensus process involving hundreds of administrators, security managers, and information security professionals, and they address both security fundamentals and awareness and the in-depth technical aspects of the most crucial areas of IT security. www.sans.org

IN-DEPTH EDUCATION AND CERTIFICATION

During the past year, more than 17,000 security, networking, and system administration professionals attended multi-day, in-depth training by the world's top security practitioners and teachers. Next year, SANS programs will educate thousands more security professionals in the US and internationally.

SANS Technology Institute (STI) is the premier skills-based cybersecurity graduate school offering master's degree in information security. Our programs are hands-on and intensive, equipping students to be leaders in strengthening enterprise and global information security. Our students learn enterprise security strategies and techniques, and engage in real-world applied research, led by the top scholar-practitioners in the information security profession. Learn more about STI at **www.sans.edu**.

Global Information Assurance Certification (GIAC)
GIAC offer more than 25 specialized certifications in the areas of incident handling, forensics, leadership, security, penetration and audit. GIAC is ISO/ANSI/IEC 17024 accredited. The GIAC certification process validates the specific skills of security professionals with standards established on the highest benchmarks in the industry. Over 49,000 candidates have obtained GIAC certifications with hundreds more in the process. Find out more at **www.giac.org**.

SANS BREAKS THE NEWS

SANS NewsBites is a semi-weekly, high-level executive summary of the most important news articles that have been published on computer security during the last week. Each news item is very briefly summarized and includes a reference on the web for detailed information, if possible. **www.sans.org/newsletters/newsbites**

@RISK: The Consensus Security Alert is a weekly report summarizing the vulnerabilities that matter most and steps for protection. **www.sans.org/newsletters/risk**

Ouch! is the first consensus monthly security awareness report for end users. It shows what to look for and how to avoid phishing and other scams plus viruses and other malware using the latest attacks as examples. **www.sans.org/newsletters/ouch**

The Internet Storm Center (ISC) was created in 2001 following the successful detection, analysis, and widespread warning of the Li0n worm. Today, the ISC provides a free analysis and warning service to thousands of Internet users and organizations and is actively working with Internet Service Providers to fight back against the most malicious attackers. **http://isc.sans.org**

TRAINING WITHOUT TRAVEL ALTERNATIVES

Nothing beats the experience of attending a live SANS training event with incomparable instructors and guest speakers, vendor solutions expos, and myriad networking opportunities. Sometimes though, travel costs and a week away from the office are just not feasible. When limited time and/or budget keeps you or your co-workers grounded, you can still get great SANS training close to home.

SANS OnSite *Your Schedule! Lower Cost!*
With SANS OnSite program you can bring a unique combination of high-quality and world-recognized instructors to train your professionals at your location and realize significant savings.

Six reasons to consider SANS OnSite:

1. **Enjoy the same great certified SANS instructors and unparalleled courseware**
2. **Flexible scheduling – conduct the training when it is convenient for you**
3. **Focus on internal security issues during class and find solutions**
4. **Keep staff close to home**
5. **Realize significant savings on travel expenses**
6. **Enable dispersed workforce to interact with one another in one place**

DoD or DoD contractors working to meet the stringent requirements of DoD-Directive 8570? SANS OnSite is the best way to help you achieve your training and certification objectives. **www.sans.org/onsite**

SANS OnDemand *Online Training & Assessments – Anytime, Anywhere*
When you want access to SANS' high-quality training 'anytime, anywhere', choose our advanced online delivery method! OnDemand is designed to provide a very convenient, comprehensive, and highly effective means for information security professionals to receive the same intensive, immersion training that SANS is famous for. Students will receive:

- **Up to four months of access to online training**
- **Integrated lectures by SANS top-rated instructors**
- **Access to our SANS Virtual Mentor**
- **Hard copy of course books**
- **Progress reports**
- **Labs and hands-on exercises**
- **Assessments to reinforce your knowledge throughout the course**

www.sans.org/ondemand

SANS vLive *Live Virtual Training – Top SANS Instructors*
SANS vLive allows you to attend SANS courses from the convenience of your home or office! Simply log in at the scheduled times and join your instructor and classmates in an interactive virtual classroom. Classes typically meet two evenings a week for five or six weeks. No other SANS training format gives you as much time with our top instructors.
www.sans.org/vlive

SANS Simulcast *Live SANS Instruction in Multiple Locations!*
Log in to a virtual classroom to see, hear, and participate in a class as it is being presented LIVE at a SANS event! Event Simulcasts are available for many classes offered at major SANS events. We can also offer private Custom Simulcasts – perfect for organizations that need to train distributed workforces with limited travel budgets. **www.sans.org/simulcast**

For group programs, please contact us at groupsales@sans.org