SANS

www.sans.org

SECURITY 503
INTRUSION DETECTION
IN-DEPTH

503.1

Fundamentals of Traffic Analysis: Part I

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

SANS

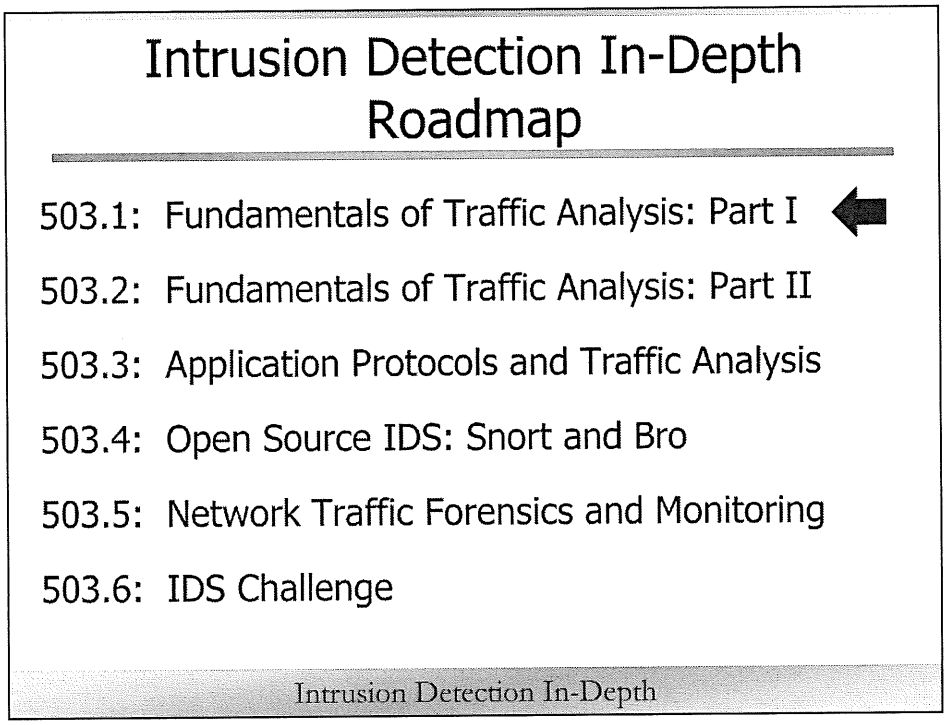
www.sans.org

SECURITY 503
INTRUSION DETECTION
IN-DEPTH

503.1

Fundamentals of Traffic Analysis – Part 1

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



Welcome to SANS Security 503: Intrusion Detection in Depth. We'll spend the next several days of this comprehensive course understanding the fundamentals of traffic analysis, TCP/IP, learning to use traffic sniffing and analysis tools such as tcpdump and Wireshark, becoming familiar with Snort and Bro, using a variety of other tools such as SiLK to analyze traffic, comprehending network forensic analysis, network architecture, and correlation. Finally, you'll put all of your knowledge to work on the last day, where you will participate in a hands-on challenge where you analyze an actual intrusion to discover why and determine how the intrusion occurred.

We hope that you find this material instructive and engaging. So, put your seats and tray tables in the upright position as we prepare for take-off!

There have been many wonderful contributors and reviewers who deserve recognition – Mike Poor, Guy Bruneau, Marty Roesch, Jess Garcia, Nathan Benson, Jen Harvey, Tim Collyer, Jesse Bowling, Vern Stark, Adrien de Beaupre, and Carrie Roberts. Dave Hoelzer and Johannes Ullrich wisely recommended some important updates and Johannes created and maintains the VM. I'd like to thank and acknowledge all of them for their assistance.

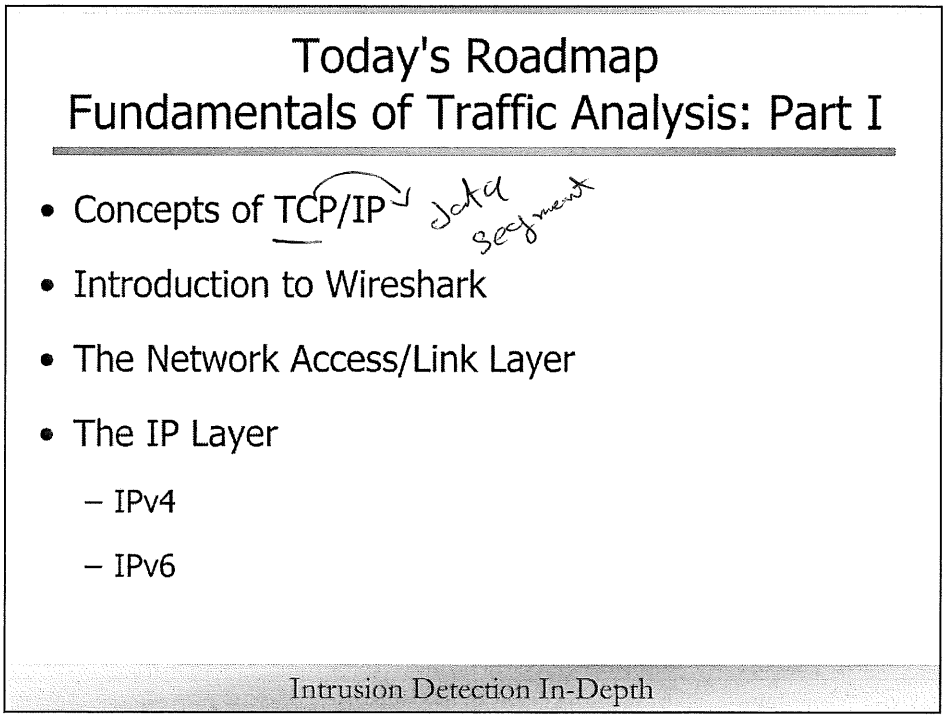
Special gratitude is extended to two amazing people who have also become friends - Sally Vandeven and Andy Laman for their major contributions of thorough review, testing labs, and insightful feedback.

Judy Novak

Fundamentals of Traffic Analysis: Part I

Intrusion Detection In-Depth

This page intentionally left blank.



Here is a roadmap for Day 1. We'll begin with some concepts of TCP/IP – network communication models, numbering systems binary and hexadecimal to view packets in raw form, and discuss the concept of "normal" in terms of protocols because you have to know normal to find abnormal.

Next, there will be an introduction to Wireshark. The purpose of this introductory Wireshark section is to get you comfortable navigating Wireshark and to learn some of its basic capabilities. We'll delve into Wireshark more later in the course, but there is a need to expose you to it early since many of the screenshots on the slides are from Wireshark output.

We'll then start to examine the lower layers of TCP/IP – the link layer, and the IP layer. We'll cover both IPv4 and IPv6.

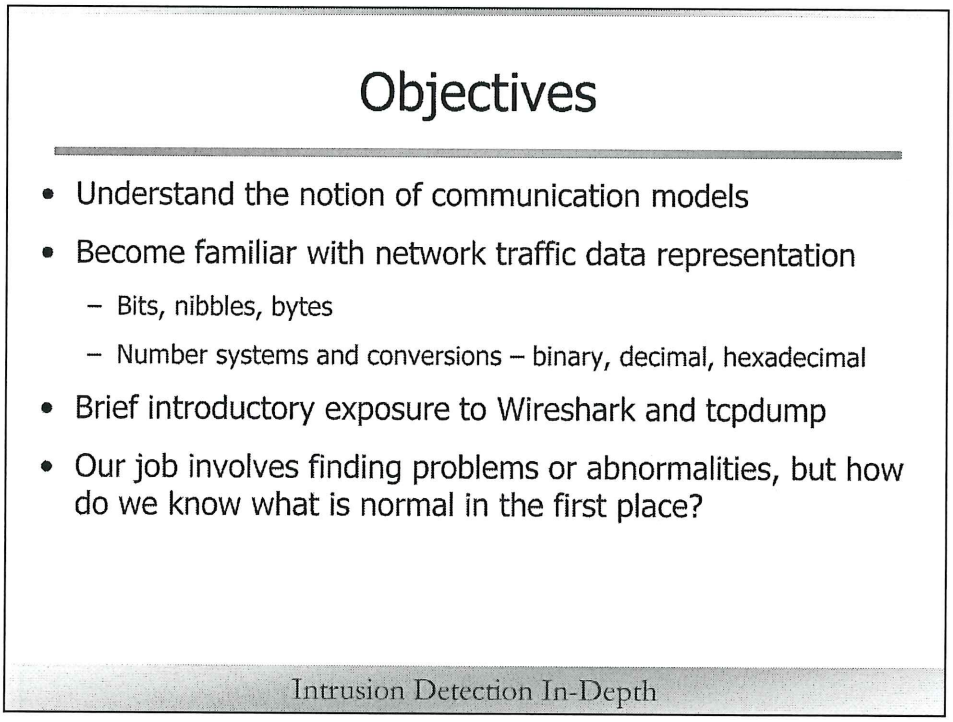
We will be covering many tools and products today. We would like to cite the author or vendor of each in advance and give credit to them for their contributions.

Tool	Vendor/Author
tcpdump	Van Jacobson, Craig Leres, Steven McCanne, Michael Richardson, Bill Fenner
Wireshark	Gerald Combs, CACE Technologies
VMware	Vmware, Inc.
Snort	Marty Roesch, Sourcefire, Inc.
Packetrix	The Flamboyant Mr. Mike Poor
Firefox	Mozilla Corporation and Mozilla Foundation
Internet Explorer	Microsoft, Inc.
Safari	Apple, Inc.
arpwatch	Lawrence Berkeley National Laboratory
ISIC	Mike Frantzen, Shu Xiao
nmap	Fyodor Vaskovich aka Gordon Lyon
Ping O' Death	Unknown
Teardrop	Unknown
OpenBSD IPv6 mbufs attack	Core Security Technologies

Concepts of TCP/IP

- Concepts of TCP/IP
- Introduction to Wireshark
- The Network Access/Link Layer
- The IP Layer
  - IPv4
  - IPv6

This page intentionally left blank.

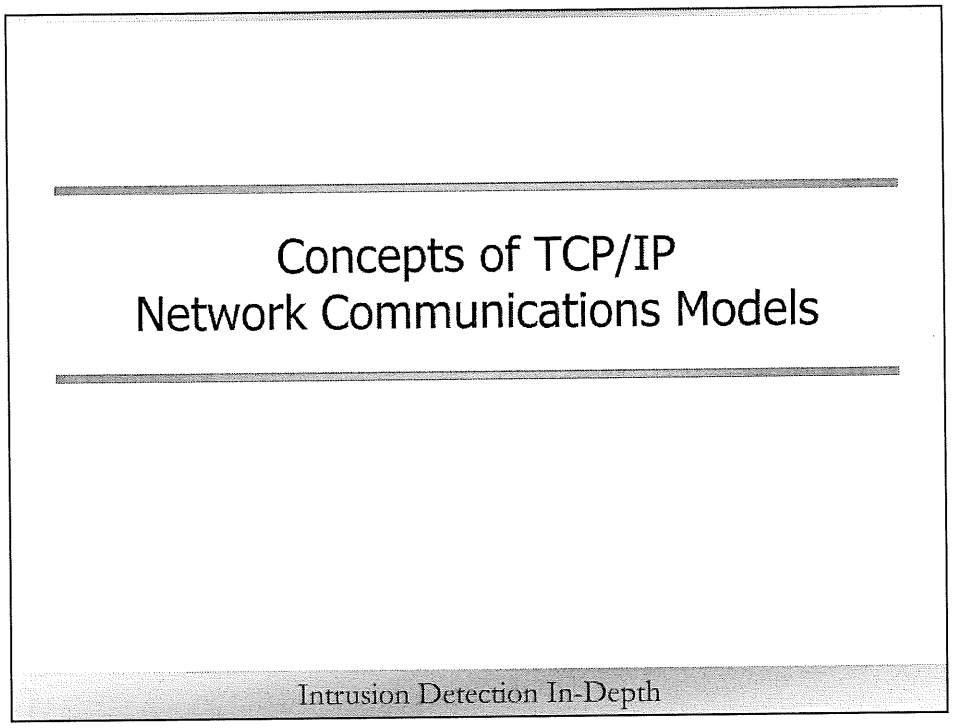


This section introduces you to the foundations required to perform traffic analysis for intrusion detection/prevention. You can rely exclusively on the interpretations and alerts from your Intrusion Detection System (IDS) or Intrusion Prevention System (IPS) to protect your network. But, if you've run either an IDS or IPS you know that it is not wise to believe everything that it tells you. Notably, there is the issue of false positives, where your IDS/IPS generates an alert that does not represent malicious traffic. For instance, if your IDS/IPS warns you of a malicious attack for an Apache web server, yet you do not run one, the alert is a false positive. But, you have to understand what the alert is about and be able to pursue the issue based on your ability to analyze traffic and packets. This pertains to false negatives too where you do not get any indication of malicious traffic. If you are informed by some other tool, for example anti-virus, that there has been malicious activity for which you believe your IDS/IPS provides protection, you'll need to examine both the traffic and IDS/IPS signatures and configuration to understand the issue.

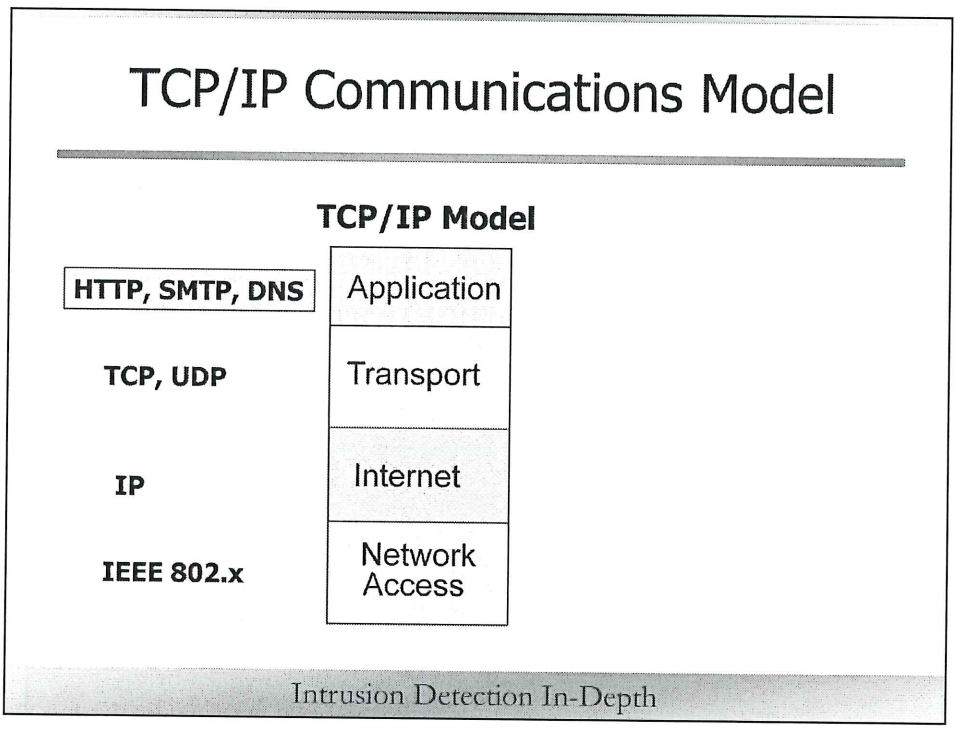
We'll cover the communication model upon which TCP/IP is based—including the notion of layering and encapsulating other layers of the network stack. This is important because it dictates how network traffic contains a standard format and order, based on these models. Another topic of great value is understanding how packet data is represented, the terminology associated with the representation, such as bits and bytes, and the ability to know different numbering systems – binary, decimal, and hexadecimal – and be able to convert values among these numbering systems with relative ease.

Next, we'll very briefly cover an overview of two of the main tools that will be used throughout the class to analyze and parse traffic – namely tcpdump and Wireshark, two open source tools. And, before we begin to attempt to find abnormalities or signs of malice, we must know where to go to try to discover what is normal and expected in different protocols.

Note: Throughout the course, the terms packet and datagram are used interchangeably.



This page intentionally left blank.



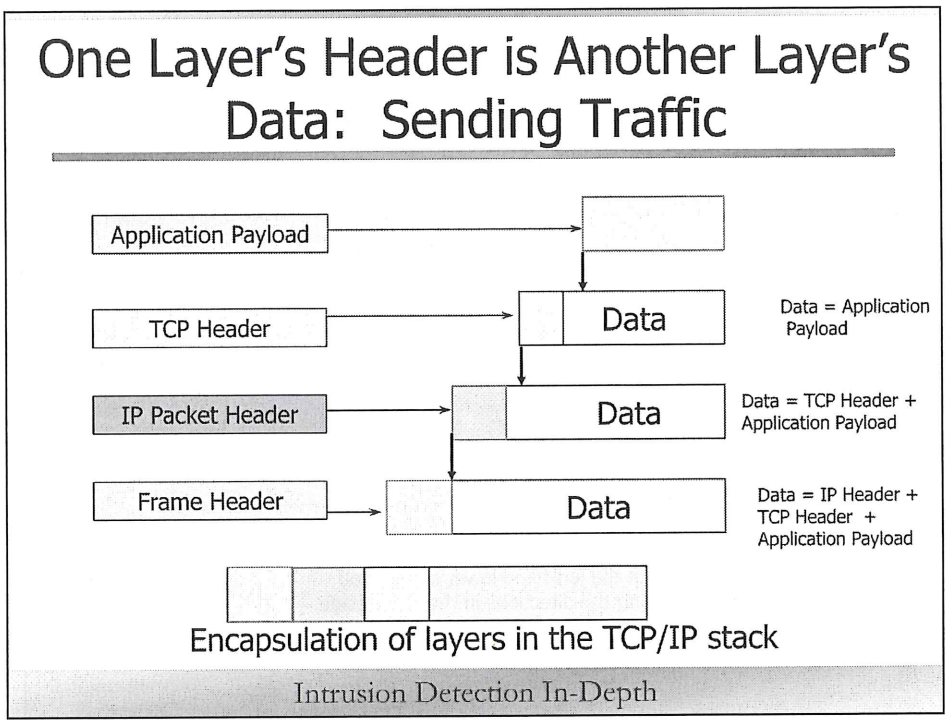
We must understand the concept of network communication models because network traffic is created, formatted, and disassembled based on these models. Frames, packets, headers, and data emulate the layers of the network models.

Conceptually, years ago, Charles Bachman on Honeywell Information Services in conjunction with the International Organization for Standardization came up with a proposed standard of how networked hosts would communicate using what is known as the OSI (Open Systems Interconnection) Model. It was offered as a universal communication standard. This model has seven layers of different functions that are to be performed by the communicating hosts. This is the origin of a given function being referred to as a "layer."

The OSI model did not gain as much favor as another model - the TCP/IP model. The TCP/IP model is conceptually similar to the OSI model, however it contains four layers instead of seven. We'll examine the concepts associated with each of these layers in the next several days.

A communication model is important for segregating the functions that are required for communications. This modularizes the concepts and hence the implementations of them. Hosts transparently communicate with each other at peer layers. In reality, though, layers talk to above and below layers to pass or receive part of the total message to be sent. In essence, we have what has been commonly known as a stack.

You'll no doubt hear the term "layer" when discussing networks. If you hear someone say that a particular piece of software/hardware operates at Layer 3, they are referring to the Internet layer. Typically, when someone talks about a layer, it is in reference to the older OSI model. For instance, Layer 5 is the session layer or more commonly Layer 7 is the application layer. Throughout the course, communications and associated functions are referenced as a 4-layer TCP/IP module. We'll try to avoid referencing by layer numbers since this may lead to confusion.



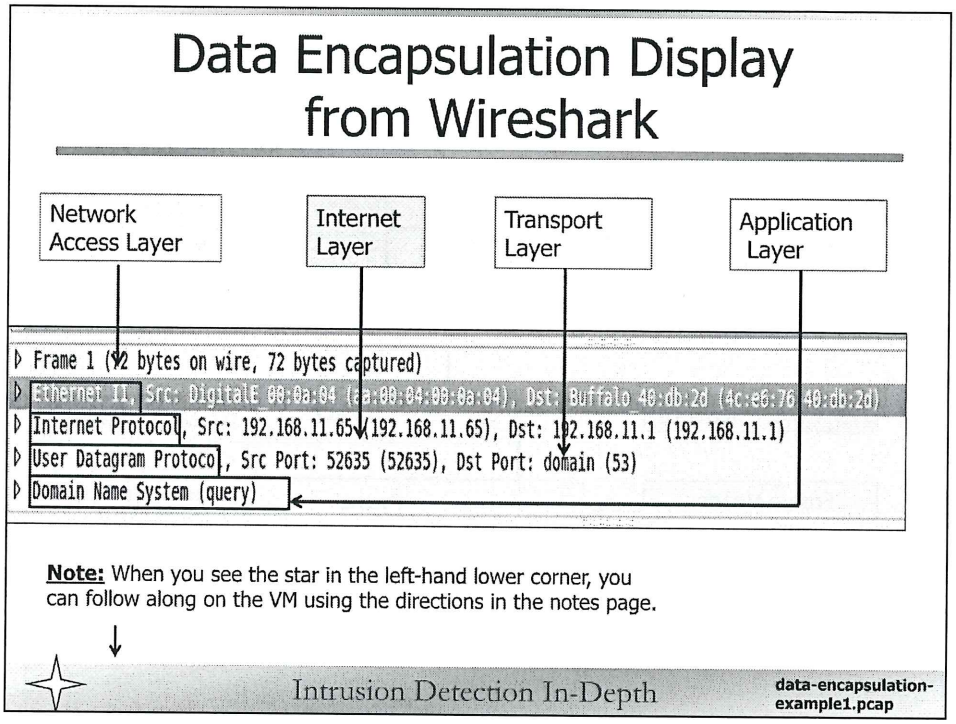
We see how the TCP/IP model works in this slide. When some type of data is packaged by the sending host's TCP/IP stack, each lower layer in the stack adds something to this message.

Starting at the top of the TCP/IP stack, the application layer is the one that supplies the payload or the data that appears after all the encapsulating headers. The application layer passes the payload down to the transport layer. In this slide, we are relying on a transport layer that involves TCP. The transport layer - in this case, TCP, adds a TCP header to the application data. It provides fields such as the source and destination ports as well as many fields which ensure that the TCP segment is reliably delivered.

Once the transport layer adds its header, it passes the message constructed so far down to the Internet layer. The Internet layer adds its own IP header to be able to deliver this packet to the correct destination IP by advancing hop to hop. The TCP header melds into what now becomes the IP data because as far as the IP or networking layer is concerned, it doesn't distinguish between the TCP header and the application data. It has no need to be concerned about destination ports or other transport layer data nor does it need to consider data payload.

Finally, the IP packet is passed to the network access layer. The network access layer supplies a frame header to the IP packet that contains information such as source and destination Media Access Control (MAC) addresses. And, the IP packet with all its "data" consisting of the IP header, transport header, and data is injected into frame data.

The process of adding header information as a packet is passed down the TCP/IP stack is known as encapsulation. When the destination host receives the frame, it has to strip off headers and pass the message to the appropriate upper layer. It essentially reverses the encapsulation process with a process of its own known as de-encapsulation.



Let's look at a unit of traffic that has traversed the network. We've captured it and displayed it using an open-source tool called Wireshark. We'll discuss Wireshark in great detail, but for now, we're simply using it to see how its display mimics the concept and format of layering. Specifically, we examine a DNS query that we've captured.

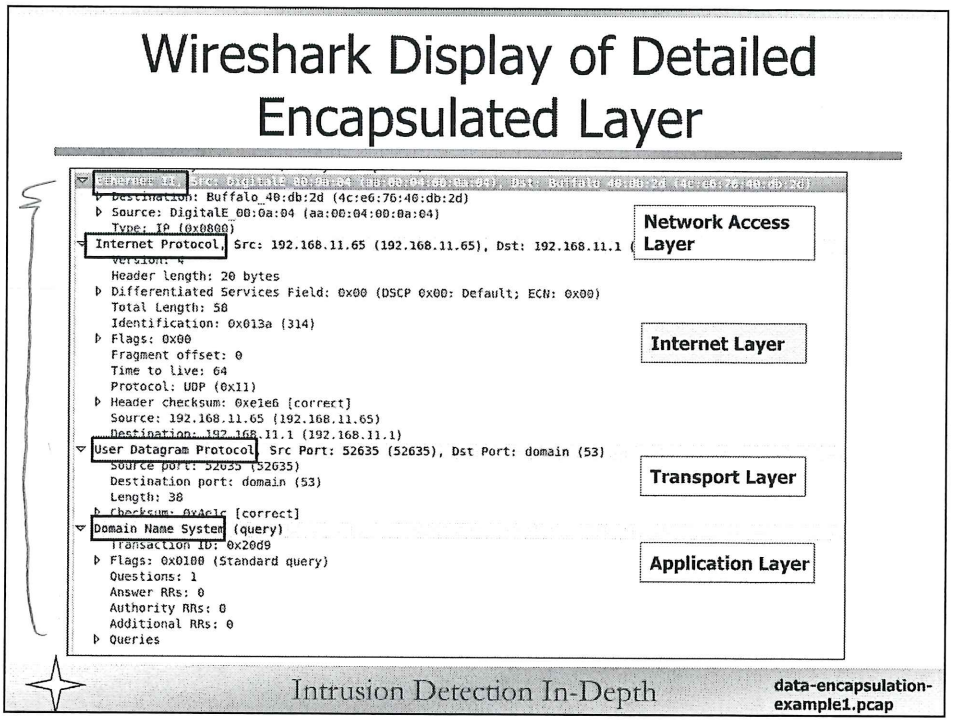
Wireshark's illustration of layers inverts the depiction we've used of the TCP/IP models. The lower layers are displayed in Wireshark first followed by layers that are theoretically above. This may take a little inverted thinking to view the layers as they are meant to be.

At the lowest echelon, we have the network access layer, also known as the link layer. The first line of Wireshark output is labeled "Frame", but this contains some summarized metadata, not the actual frame (network access layer) header itself. That is found in the next line under "Ethernet II". Wireshark displays the source and destination MAC addresses. A standard MAC addresses is a 6-byte unique designation for the network interface card. This value is assigned at the factory where the card is manufactured and is not alterable. We'll discuss MAC addresses in more detail in later.

✦ Open a terminal on the VM to enable you to enter the commands to view demonstrations.
Day1 demonstration pcaps are found in /home/sans/demo-pcaps/Day1-demos on the VM.

Navigate to the proper directory:
cd /home/sans/demo-pcaps/Day1-demos
Next open Wireshark with the pcap for the demonstration. Enter the following on the command line:
wireshark data-encapsulation-example1.pcap

Next, Wireshark logically displays the IP layer that follows the frame header. The transport layer – in the above output UDP – logically follows the IP header, again since this is the order in the frame. Wireshark shows the source and destination ports in its summary line. The destination port is 53 that is usually associated with DNS. Finally, a brief description of the application payload – a DNS query is displayed last.

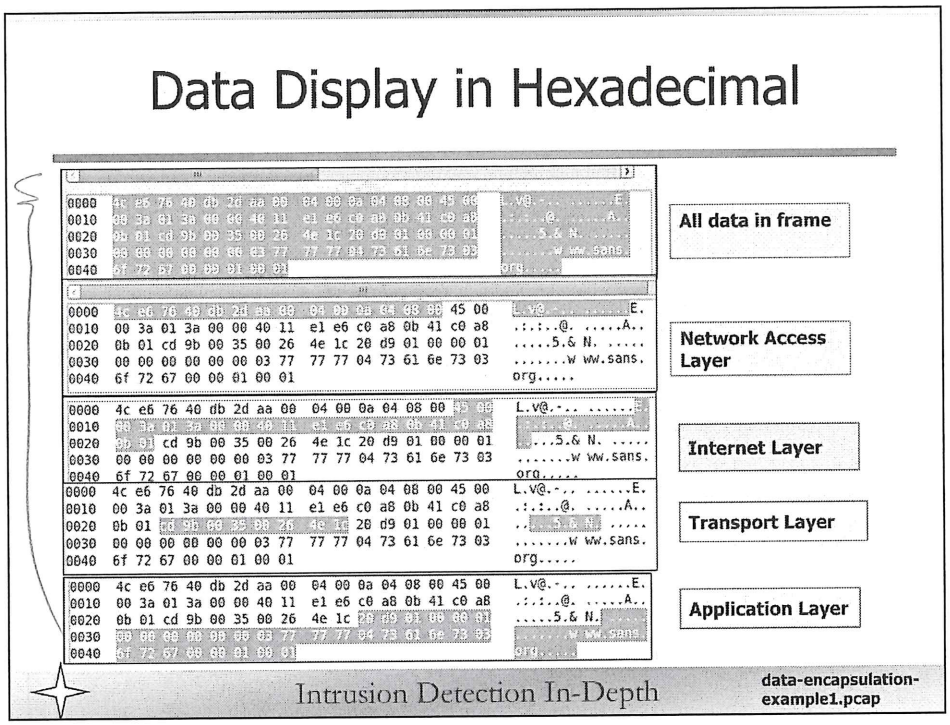


packet detail pane

This is a more detailed look at the fields and values found at each layer of the DNS query. Wireshark is able to expand the detail associated with a given layer or multi-value field by clicking on the right facing triangle to the left of an expandable field. We see Wireshark's interpretation of the frame. This is an important point to stress for both Wireshark and tcpdump – what is displayed is an interpretation only by the tool of the fields and values. Most times the tool – either Wireshark or tcpdump – will display an interpretation that is meaningful to you and is an accurate representation of the bits and bytes in the actual frame, packet, header, or payload. Yet, there are instances where the interpretation is confusing, wrong, or incomplete.

Bear in mind, though, that the most accurate way to validate that the output is correctly rendered is to examine the bytes in the layer or field of interest. These values are represented in hexadecimal. If you were to validate the translation, you would need to know the layout of the given protocol that you are examining and how to convert the hexadecimal values into a more meaningful decimal representation. Don't worry, you will get plenty of practice decoding headers and protocols and making sense of the associated values!

Expand each of the layers (Ethernet II, Internet Protocol, User Datagram Protocol, and Domain Name System) by clicking the right facing triangle to the left of each layer name.

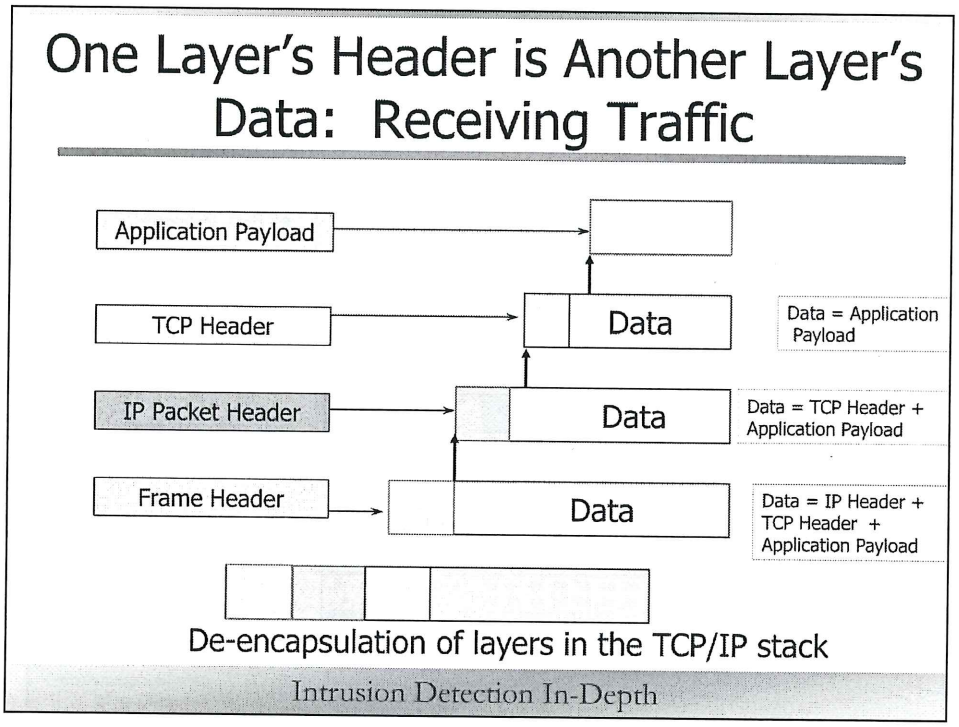


Let's drive home the point of hexadecimal interpretation of a given protocol(s). Perhaps you don't want to rely on a tool's interpretation, perhaps no protocol dissector exists to interpret a protocol, or perhaps you are just curious how the interpretation is actually performed.

This slide depicts how Wireshark performs interpretation by layer. We've honed in on a particular portion of the Wireshark display known as the packet bytes pane. We'll discuss more about Wireshark output panes when we learn to navigate and use Wireshark. This is the bottom pane of the Wireshark display and it displays the hexadecimal values that are found in the frame, packet, header, payload, etc. The top packet bytes pane is the hexadecimal output of the entire frame. There is no notion of layers in this particular output; it is presented above to give you the template frame of hexadecimal values from which the layers shown below it are derived.

In the previous slide, Wireshark's packet detail pane was shown. This is Wireshark's more human readable interpretation of a particular layer or protocol of interest. When we clicked on the "Ethernet II" network access layer using the expandable arrow, we saw the packet detail pane. Concurrently, the bytes pane highlighted in blue appeared. These are the hexadecimal bytes in the frame associated with the network access layer exclusively. This is repeated for each of the other layers in the frame to highlight the hexadecimal output associated with them.

The point here is that we are now looking at the hexadecimal representation of the layers. We've examined the layers as theory, yet here the concept is manifested as hexadecimal data depicted as encapsulated layers of the TCP/IP model. The data following any one of the above layers is payload data to the given layer. For instance, if we examine the IP header beginning with hexadecimal 0x45 and ending with hexadecimal 0x01, all the bytes that follow that represent the UDP header transport layer and the DNS query are considered data for the IP layer. As you can see in the rightmost column of data, Wireshark renders printable characters in ASCII.

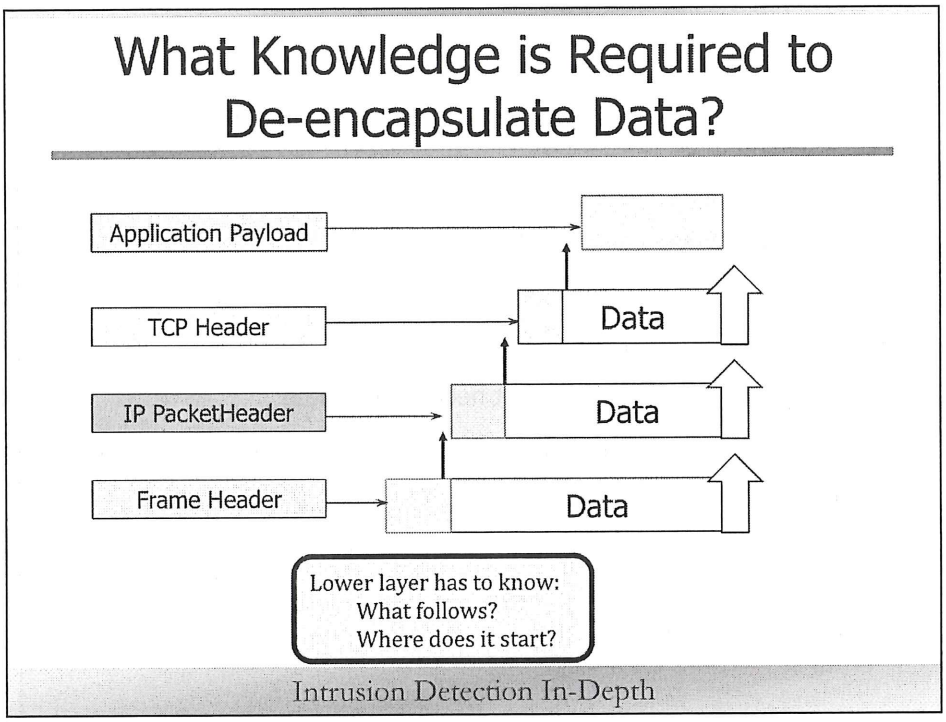


The reverse process of encapsulation occurs when traffic is received. It has to be interpreted by the receiver and this process is performed using something known as de-encapsulation.

A frame arrives at the receiver. The network access layer gets the frame, analyzes the data in it, figures out where the IP layer begins, strips off the frame header, and passes the IP header and everything that follows it as data up to the Internet layer. The Internet layer performs a similar analysis, strips off the IP header and passes the remaining data – transport header and what follows to the transport layer. The transport layer analyzes the header, strips it off and passes the remaining data up to the application layer.

Each layer processes the data that is associated with it and nothing more or nothing less. This layer independence permits modifications of a particular protocol by those who define the standards, without affecting any other layer.

The process of de-encapsulation sounds simple enough. Though, there must be some information in each layer that facilitates the process. This is discussed next.

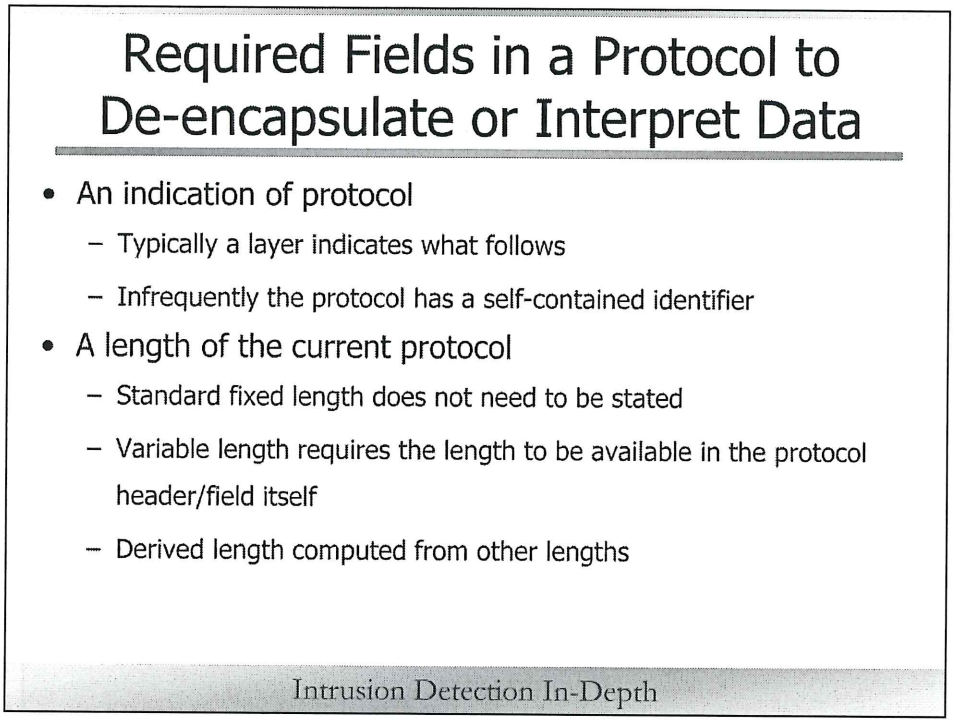


If you think about it, a chunk of data eventually arrives at the network access layer via a network access card. From there the receiving TCP/IP stack must properly interpret the header and data portions at each layer. How exactly does this happen? First, the lower layer must know its own format and which upper layer to pass the data for appropriate translation. The upper layer inherently knows its own format and how to make sense of it.

Another important data question is where does the current layer's header stop and the upper layer's data start? Some layer headers, such as an Ethernet link layer header are a fixed size and need not designate that size in the header. This is known as an implied length as it goes unstated. Other layers may have a variable length header or portion of the header. Any data in the header that is variable length must have an accompanying length to delimit one field from the next. For instance, both the IP and TCP layers have optional data that can be supplied in the header. Therefore, both have a header size that reflects the number of bytes in the header to indicate whether or not optional data is present.

There are some portions of the IP and TCP headers (the options fields in each that can carry one or more options) that supply the option code for each option present. Unlike layer decoding, this does not indicate the option code that follows; rather, it identifies the current option. This is self-identifying, rather than pre-identifying the layer that follows. Since there can be one or more IP or TCP options, each designates it own length for determination of where the current option ends and the next begins.

Finally, there are lengths that are derived, notably the size of the TCP data. We'll examine this notion in a few slides.



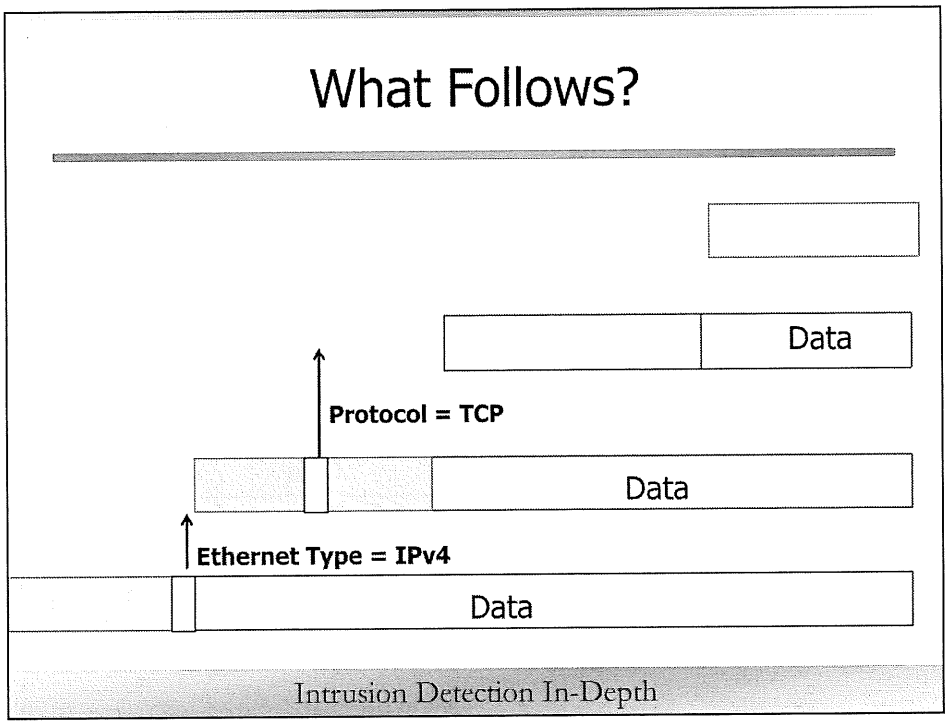
Let's examine the required fields in any given protocol to de-encapsulate layers, and let's extend the concept to include interpreting data within a layer. As we've seen, there must be some identifier of a protocol. Usually this can be found in the previous layer. At times, there is an identifier within a given field of a protocol. For instance, IP and TCP options field may contain one or more options. Each option has a field that identifies the option type. It is self-contained rather than being derived from the previous layer.

And, there has to be a means of knowing where the given layer or field within stops and another one begins. This is accomplished via a length value. When a protocol or field is a standard fixed size, there is no need to include it in the protocol itself since it wastes bytes. If a protocol or field may vary, a length is required. For instance, there may be optional TCP values after the standard TCP header, necessitating that a header length be stored in the header.

Finally, there can be a derived length. There is a length value in the IP header for the length of the entire packet. Yet, you'd think that if every subsequent protocol had some indication of size this would be unnecessary. There is no length associated with the data that follows the TCP header, as an example. But this can be computed as we'll see.

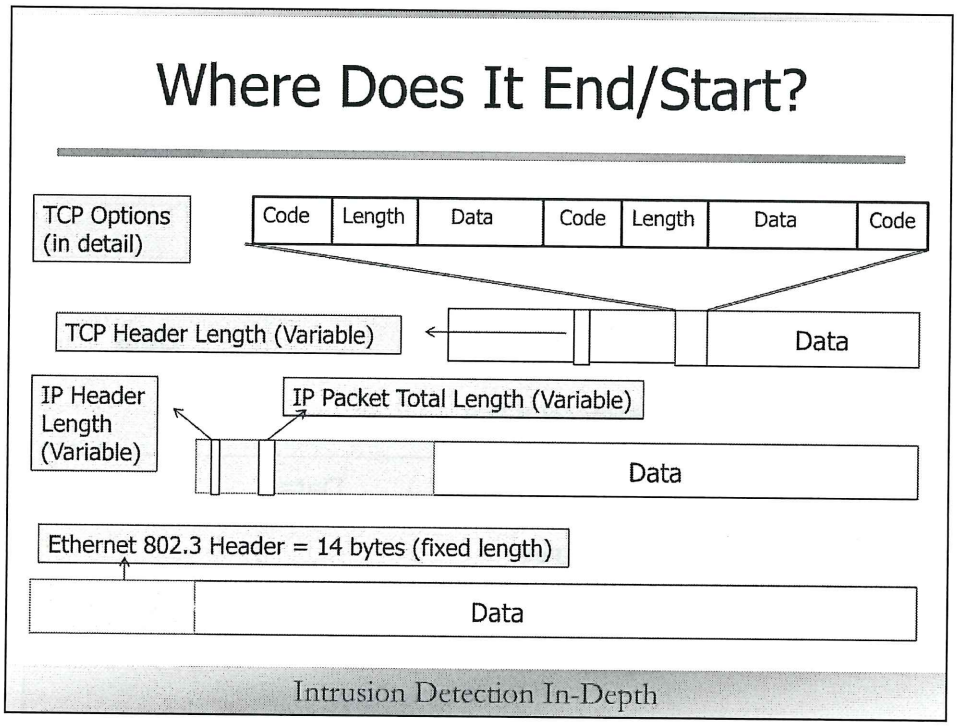
Perhaps you are thinking to yourself, "Why do we need to go into such detail about how to disassemble a given protocol and what follows? After all, don't we have Wireshark and tcpdump to do this?" Indeed they are excellent tools and we anticipate that you will rely on them most of the time. However, we feel it is an invaluable skill to be able to view a packet in hex and figure out the basics of it – what protocols are contained and where they are located in the frame or packet.

This is helpful for a couple of reasons. First, you cannot always rely on the accuracy of Wireshark and tcpdump – remember they make interpretations only of protocols. It's infrequent when they get something wrong, but it happens on occasion. And, perhaps there is a new protocol that doesn't have a dissector. This requires you to find some documentation on it and be able to disassemble or reassemble it yourself. Comfort and familiarity with the process will make it easier.



Now, let's follow a particular frame through the process of de-encapsulation. The network access card supports a given link layer protocol such as Ethernet or wireless. It inherently knows how to interpret that particular link layer. This means it knows the format and layout of all fields in the frame header. We'll use Ethernet 802.3, one of the most common link layers in this example. There has to be some indication of what protocol follows the Ethernet header. For instance, it can be IPv4 or IPv6. The data must be passed to the appropriate IP layer software to handle. This is determined by a type field in the Ethernet header.

Let's say that we are dealing with IPv4. The Ethernet layer passes all data following the Ethernet header to the code that processes IPv4. The IPv4 header protocol field designates the transport layer that follows. The transport layer has no notion of the application that follows, nor of the protocol format associated with the application. For instance, suppose you have a TCP header that contains a destination port of 80. We may assume that this is HTTP, but it does not have to be. The application layer, HTTP, is correctly parsed because the receiving host has application software that listens on a given port and knows what application it is and the application's data format.



Ethernet 802.3 is a standard fixed 14-byte length so there is no need to include a length field and consume space in the header. Therefore the data following the Ethernet header begins 14 bytes after the Ethernet header. This is the data that is passed to the IP layer.

Let's say that we are dealing with IPv4. The Ethernet layer passes all data following the Ethernet header to the software that processes IPv4. An IPv4 standard header is 20 bytes long, however, there can be IP option data causing the IPv4 header to expand to a variable length up to 60 bytes long. The IP header has a field that contains the header length so it knows where the data that follows the IP header begins, enabling it to pass it to the transport layer. The IP header has a field for the entire IP datagram length. For some protocols, such as UDP, specifying the IP datagram length is unnecessary (yet still specified) since it can be derived. The IP header indicates its length and the UDP header length includes the UDP header size plus the UDP data. But, TCP and ICMP do not include a length for the data that follows. Without this, there is no way to validate the size of the data and ensure that the length of the packet matches the expected length. After all, it is possible for someone to craft a malformed packet or for a packet to get corrupted in transit, requiring validity checks.

The transport layer header that follows the IP header may be a variable length with options as with TCP, or it may be a fixed length as with UDP and ICMP. This allows the transport layer to strip off the appropriate data and pass it up to the application layer. Let's look at TCP as an example. It has a TCP header length value, delimiting where the header stops and the data begins. But, how much data follows the TCP header? This is a derived length, the formula is:

Total IP datagram length – IP header length – TCP header length = data length

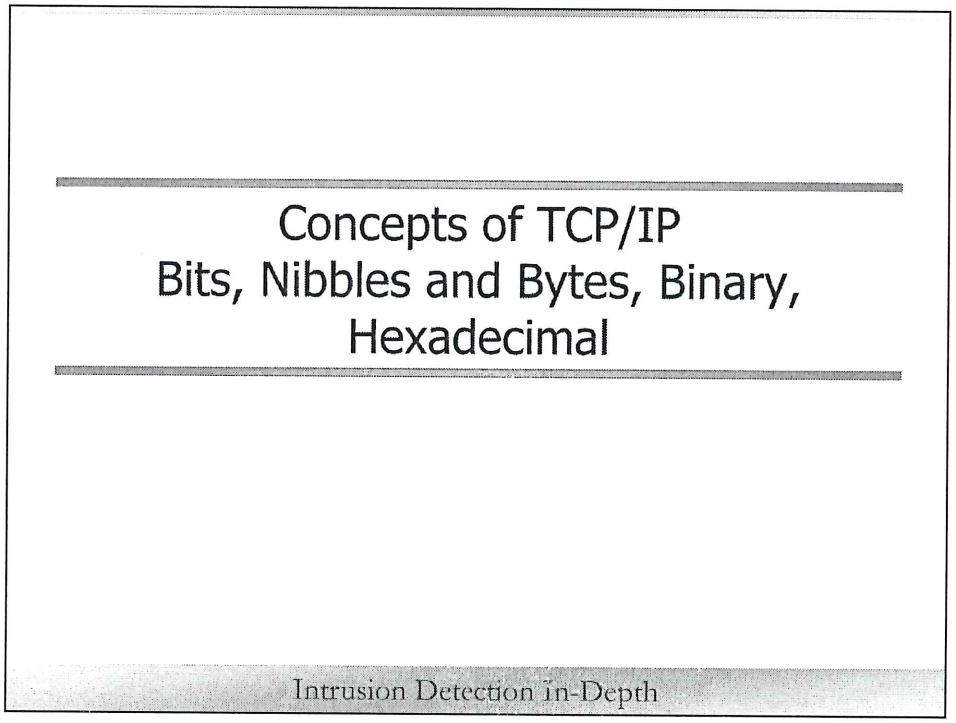
Let's use a real example. Suppose a packet has an IP header length of 20 bytes, a total IP datagram length of 1500 bytes, and a 40-byte TCP header - 20 bytes of standard header and 20 bytes of TCP options. Using our formula:

1500 – 20 – 40 = 1440

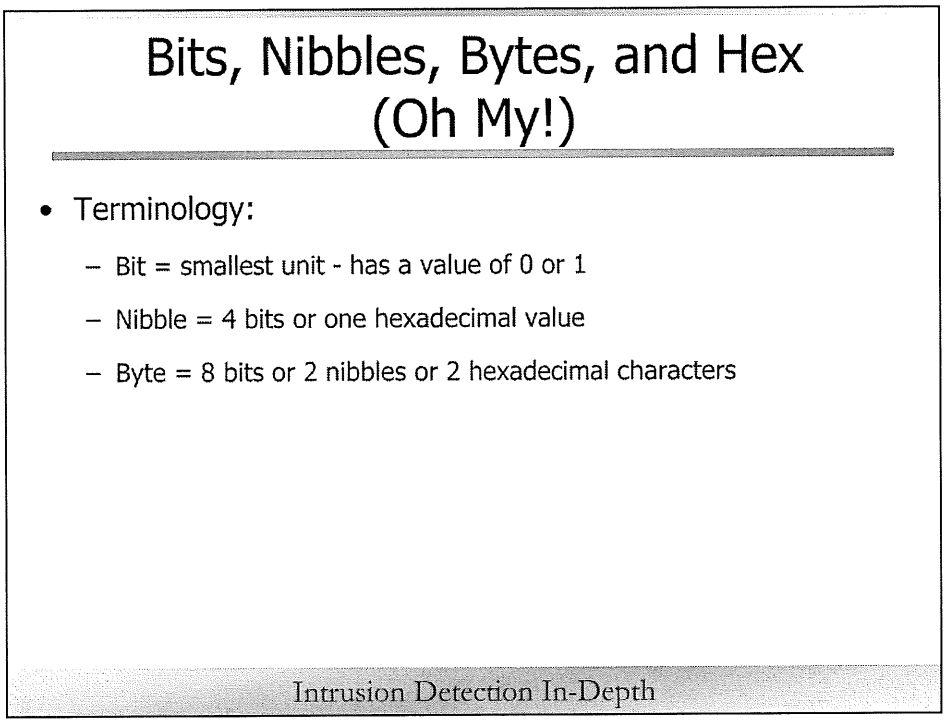
1440 bytes are expected to follow the TCP header.

Finally, let's examine the format of the TCP header options. This same scheme is used for IP options as well. As mentioned, there can be multiple TCP options. Any IP/TCP option that is greater than a single byte adheres to the format of an option code, an option length, and option data. We have a self-identifying option code – it identifies the current option, not the next one. The length includes the data, and the option code, and the length field itself. There are two IP/TCP options – EOOL, also known as EOL (end of options list 0x00) and NOP (no operation 0x01) that are self-identifying. They consist of the option only, no data, so there is no need for a length field. You may be wondering the purpose of these two one-byte options. IP/TCP options must fall on a 4-byte boundary. If they naturally do not, either one or more NOP or EOL option must be used to pad to the 4-byte boundary. The EOL, if used, should be the last option. The NOP can appear anywhere in the IP or TCP options to pad a single or a subset of options to the 4-byte boundary.

NOP is used for padding

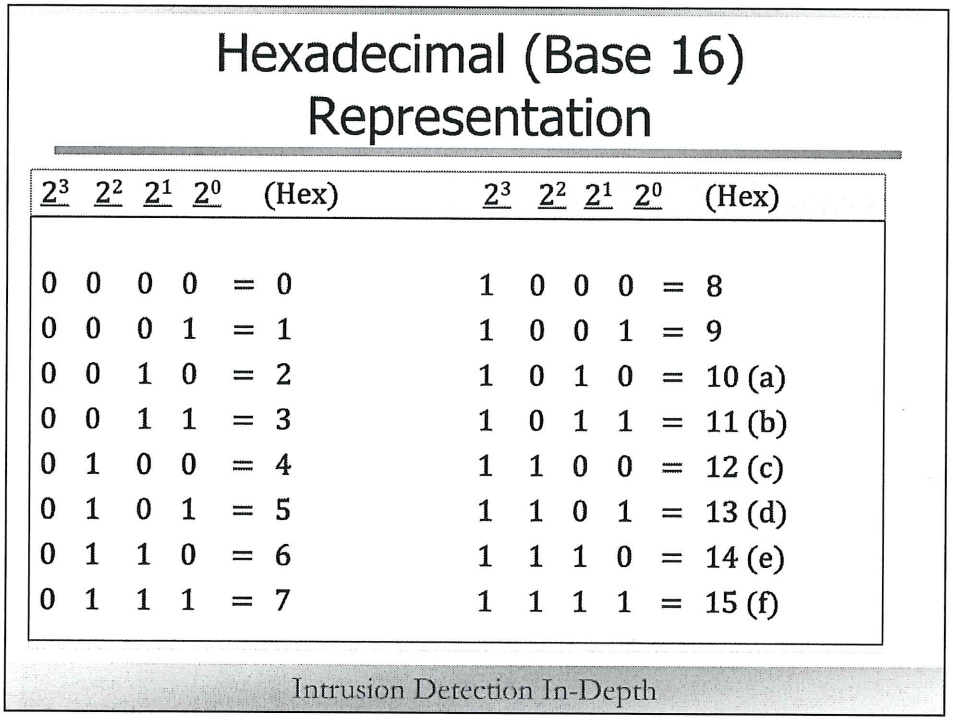


This page intentionally left blank.



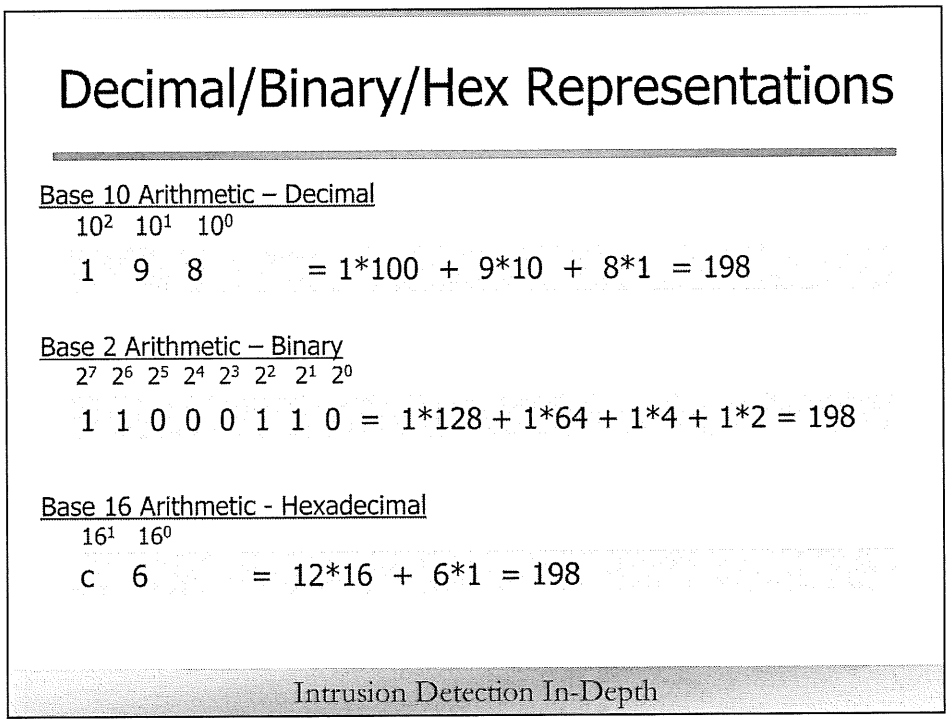
Ultimately, in the next few slides, the goal is to introduce you to how to make sense of all that hexadecimal output used by many tools like Wireshark and tcpdump. But first, let's make sure you understand the concepts of network data representation. All data is represented as a single or series of binary values – either 0 or 1 – known as a bit. This is the smallest unit of representation. Often times we will transpose bit representations into a more succinct format, and ultimately more readable, known as hexadecimal, hex for short. But, let's first examine bit values. Bits are actually represented as powers of base 2, also known as binary.

Another term used is a nibble. This is half a byte (hence the comical label of nibble), or 4 bits long. We'll see in the next slide that 4 bits are typically represented using hexadecimal notation because binary is way too long and unwieldy. The final unit of interest is the byte. It is 8 bits, or 2 nibbles, or 2 hexadecimal characters.



The output in this slide shows you all the values in the hexadecimal numbering system. When representing hexadecimal, we have a numbering system that goes from 0 to 15. The problem comes in representing values above 9 in a different scheme so that we can differentiate decimal and hexadecimal. A value of 10 decimal is a different value than 10 hexadecimal. A value of 10 hexadecimal has a value of 16 in decimal.

So, when we get to values above 9, we use letters to represent 10 – 15 as you can see in the values in parentheses above. Since hex and decimal representations are both used and may be confused, whenever you see the notation of "0x" before a value, this means that it is a hex value. For instance, 0x32 is the hexadecimal characters "32". The decimal value of 32 is different from the decimal value of 0x32.

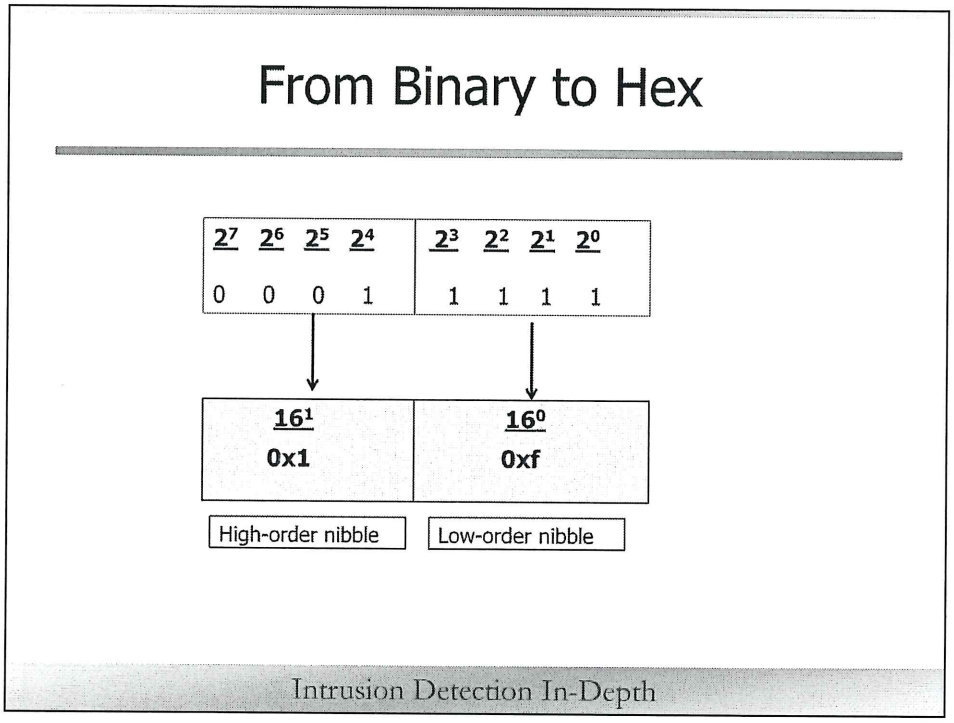


When we are dealing with packet data we need to be able to effortlessly shift and translate among different numbering systems. The numbering system with most familiarity to us is base 10 or decimal. The number 198 has intrinsic meaning to us because we are so used to dealing with it. However, when you break it down into its base 10 components, you represent each of the digits as increasing powers of 10 from right to left. As you see, we arrive at $(1 * 10^2) + (8 \times 10^0) + (9 * 10^1)$. Any base with a power of 0 assumes a value of 1. Easy enough?

Let's transpose that same theory to binary. You may be wondering why you need to know binary. When all is said and done, all data is represented as bits, or binary data. Most of the time we manage to avoid binary simply because it is too verbose and unwieldy to use. Instead we use hexadecimal that is really just a summarized version of binary. We primarily examine data in binary form when we are dealing with individual bit settings in a byte(s) or nibble(s). There are fields that are a bit in length, such as the IP header Don't Fragment flag. As well, there are fields such as the TCP flags where each bit in the byte represents a unique flag setting.

First, let's take a look at the decimal value 198 in binary. We begin by representing base 2 values as incrementing powers of 2 from right to left. Turning on the bits in the proper powers of 2, we can arrive at 198 again.

You'll see hexadecimal values when sniffers such as tcpdump or Wireshark display low level packet data. Decimal value 198 translation to hex uses the same theory as binary. The only difference is that we use a representation of powers of 16, and so we once again arrive at 198.

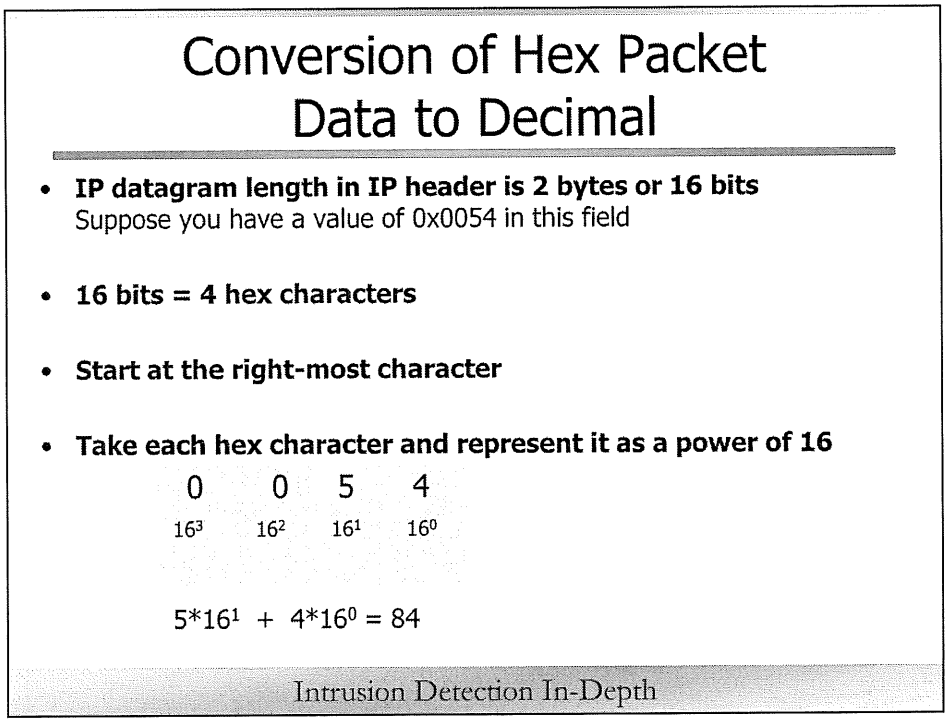


We're part way to unraveling the mystery of deciphering hexadecimal output and making sense of it. Before we continue, it's helpful if you are able to convert binary to hexadecimal and vice versa. This will come in handy when we look at hex output of a packet and need to examine a field that has many individual bit settings. We'll spend a lot of time on determining what TCP flags are set given two hexadecimal characters for this byte-long field. There are 8 different TCP flag fields, each representing a unique TCP flag. That means you need to be able to take the TCP byte value represented by two hex characters and discern which bits or flags are set.

Let's examine the binary value of the byte 0001 1111. The 4 bits that comprise the rightmost nibble of the byte are called the low-order nibble because they represent decimal values 1-15 of a byte. The 4 bits that comprise the leftmost nibble of the byte are called the high-order nibble. They have greater values based on their placement in the whole byte. When paired with a low-order nibble, the high order nibble takes on successive powers of 2 with exponents of 4, 5, 6, and 7.

The rightmost or low-order nibble is 1111 in binary. If you add up all these bit settings, you get 1+2+4+8=15. The bits in the high-order nibble (from left to right) take on the values of 16, 32, 64, and 128. Therefore, the high order nibble above has a 1 in the bit that represents a decimal value of 16. If we add this to the value of 15 from the low-order nibble, the result is a decimal 31.

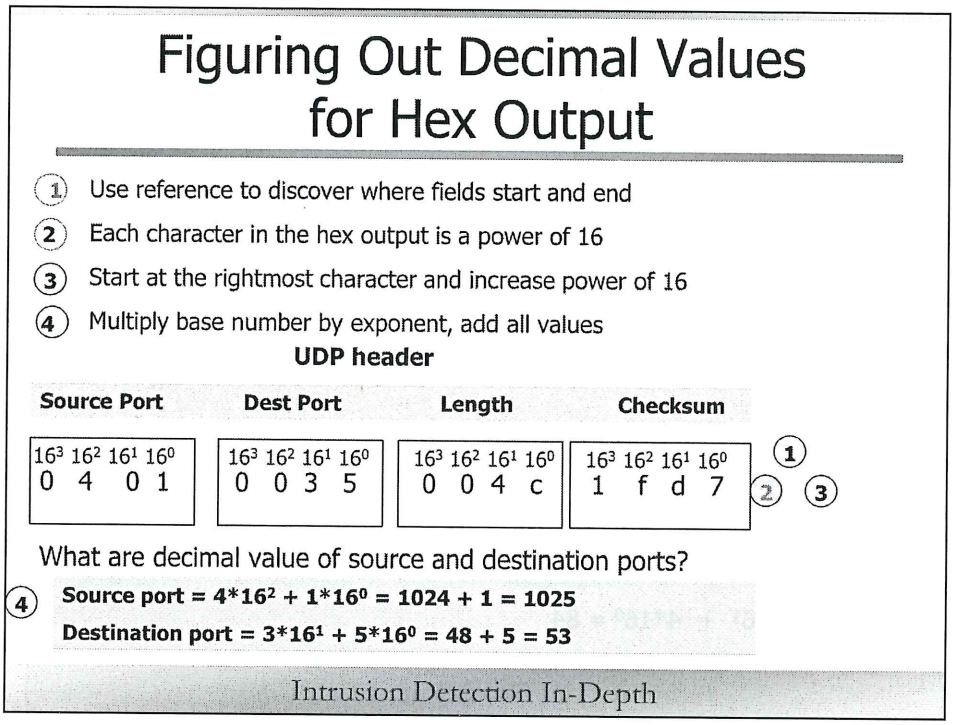
The final step is to translate the individual nibble values to hexadecimal for a more readable format. Each nibble represents a single hex character. For now, we disregard the nibble's place in the byte when computing this and examine it as a nibble with values of $2^0 - 2^3$. The low-order nibble has a value of 15 or 0xf and the high-order nibble has the value of 1 or 0x1.



Finally, we arrive at our goal of interpreting packet data represented in hexadecimal. Let's see how this is done. First, figure out where the field you need to convert begins and ends. Then start at the rightmost or least significant hex character and label that as a power of 16^0.

Simply continue labeling the remaining hex characters of the field you are converting as increasing powers of 16. Finally, after all of the characters are labeled as powers of 16, multiply the hex character by the appropriate power of 16 where it falls in the field.

In the above example, we are looking at the IP datagram length field. We have 4 hex characters because the length is a 16-bit field; because they are non-zero, we really only need to label the two right-most characters. After we do this, we find we have a 4 in the 16^0 position meaning we have 4*1 or 4. The next character of 5 is in the 16^1 position. So, we multiply 5*16 for a product of 80. Therefore, we add these two values together (4+80) for the decimal conversion of 84.

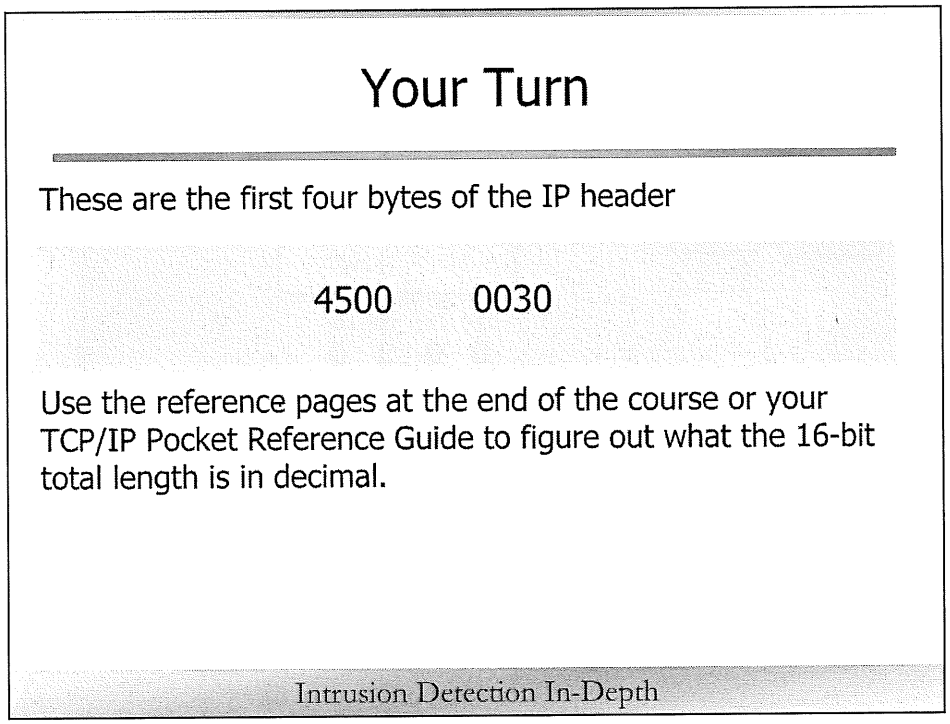


Let's assume that we are looking at a field or fields that have numeric values. In other words, we are not looking at a string payload. Let's use 8 bytes of hexadecimal output from a UDP header to describe the process of figuring out the decimal values of all the fields.

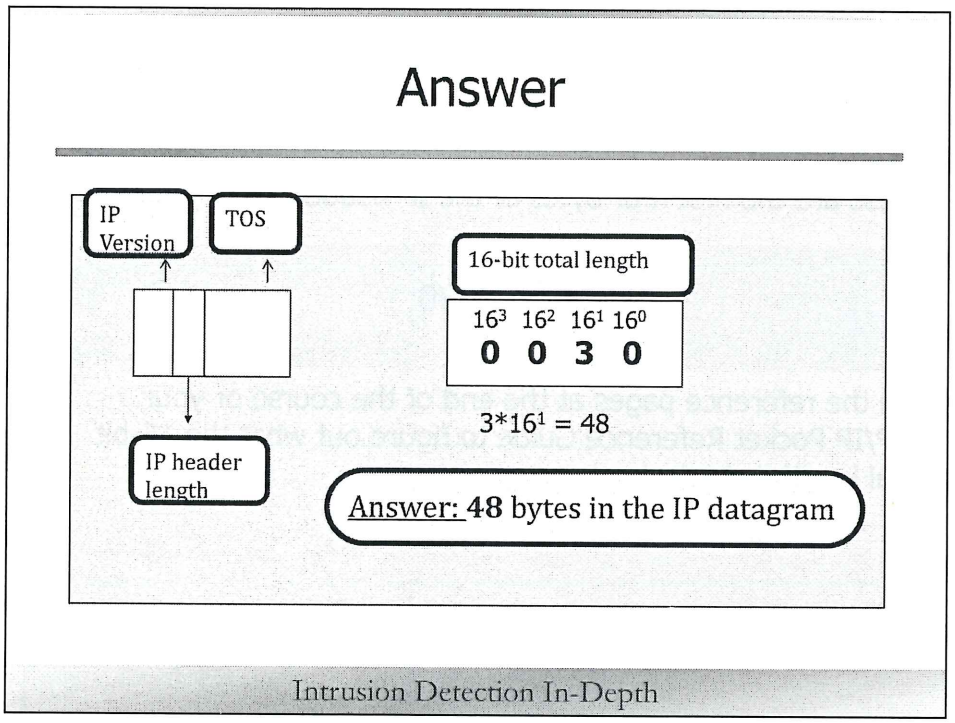
The first thing that you need to do is to identify what protocol or field that you are examining. This is the 8-byte UDP header. You'll need to use some reference, such as TCP/IP Illustrated, Volume1 by Richard Stevens or the references at the back of the course or the TCP/IP Pocket Reference Guide you received in your course materials to identify the fields in the UDP header. Remember that each character that you see in the output is one hex character (4 bits). You'll discover that there is a 16-bit source port, a 16-bit destination port, a 16-bit UDP length and a 16-bit checksum in the UDP header. Coincidentally, these are all 2 byte fields – or 4 hex characters. You see that we divide up the hex output accordingly.

Next, start with the rightmost hex character for the field in the packet that you are interested in and label it with an exponent of 16^0. For each hex character associated with the field you are examining, move left and increase the power of 16 until you hit the leftmost hex character in the field. Then, multiply the base by the exponent above it and add all the values.

Using the source port 0401 as an example of our field of interest, we start with the rightmost character (1) and label the exponent 16^0. Next, we only have one more character (4) that is non-zero with an exponent 16^2. Now, we multiply the rightmost character 1 by 16^0 and get a result of 1. Then we multiply the 4 by 16^2 to arrive at 1024. Therefore the source port is 1025. Let's move on to the destination port. There is a 5 in the 16^0 position yielding 5, and a 3 in the 16^1 position to get $3 * 16 = 48$. Our destination port is 53, the well-known port associated with DNS.



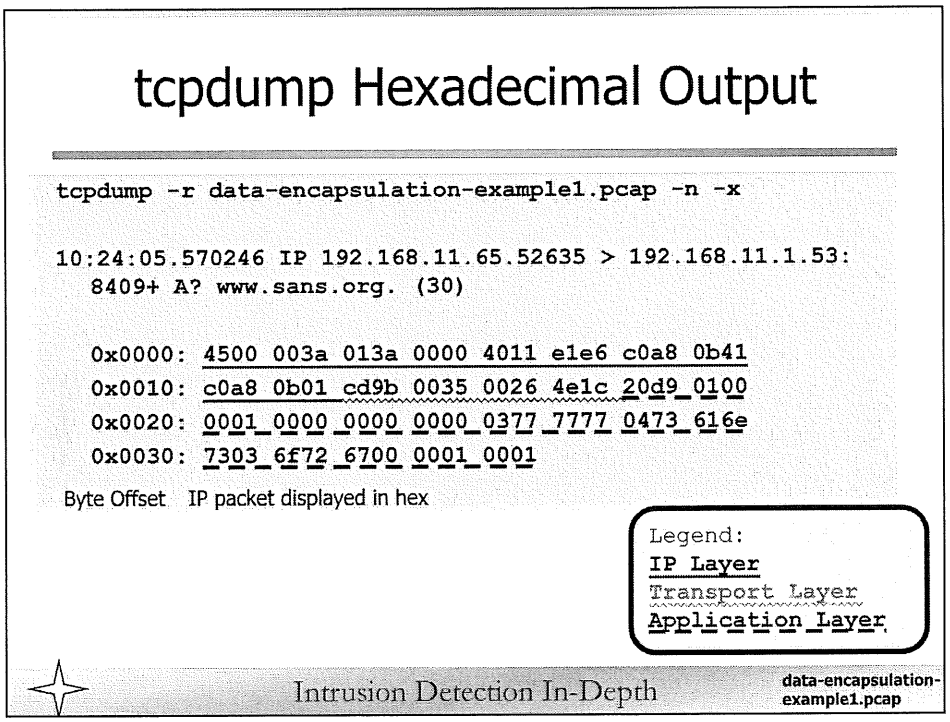
Here is your opportunity to use what you've just learned. Figure out the decimal value of the 16-bit total length. Use the reference materials that you've been given to find a layout of the IP header and where the 16-bit total length falls in the IP header. Once you've discovered that field, use the methods discussed in the previous page to figure out the decimal equivalent of the hex value.



The first thing we do is look at the layout for the IP header. When you are dealing with offset placement of a field from the beginning, counting begins at 0, not 1. This means that the 16-bit total length field is found in the 2nd and 3rd bytes offset from the IP header. We find a value of 0030 in these 2 bytes. We methodically label all the hex digits in this field as increasing powers of 16 starting at the rightmost hex character: because we only have one non-zero value in the IP length field, we really only need to figure out its value.

The non-zero value of 3 is located in the 16^1 position. So, we simply multiply 3*16 and discover that the IP length is 48 bytes.

The TOS field name has been changed to the differentiated services byte in case you see it labeled as such.



Let's analyze data-encapsulation-example1.pcap by displaying it as tcpdump hex output. Tcpdump is another tool in addition to Wireshark that we'll use often in the material. It is not nearly as pretty as Wireshark in the beauty contest of network traffic output. However, it is quite useful for a more succinct view of the data without the overhead of a GUI.

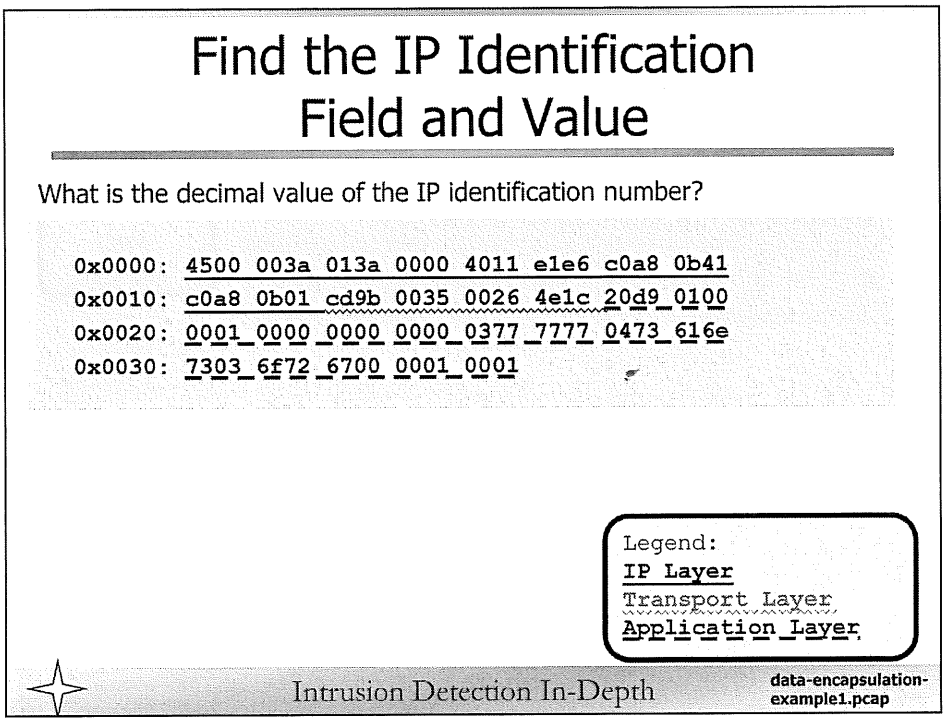
The tcpdump command has many command line options or switches to allow you to read, write, and display data in different formats. By default, tcpdump does not display output in hex. However, the command switch of –x was included to show the packet in hex. The tcpdump command used reads the filename (-r) that follows, and the –n option informs tcpdump to disable hostname and port or service resolution. Hostname resolution involves doing DNS lookups, consuming time and resources. The recommendation is to disable DNS resolution whenever feasible. The –n option also displays well-known port or service names instead of the numeric representation.

The default output of ASCII mode is displayed below the tcpdump command. The first field is the timestamp of the capture time. This is followed with "IP" indicating that is the protocol that follows the link layer header. Next, is a combination of source IP (192.168.1.65) and source port (52635) a delimiter of ">" to separate source and destination output followed by a combination of destination IP (192.168.11.1) and destination port (53). Tcpdump does some minimal DNS protocol decode. We'll talk more about DNS and its fields later in the course.

Exit from Wireshark and enter the following on the command line:

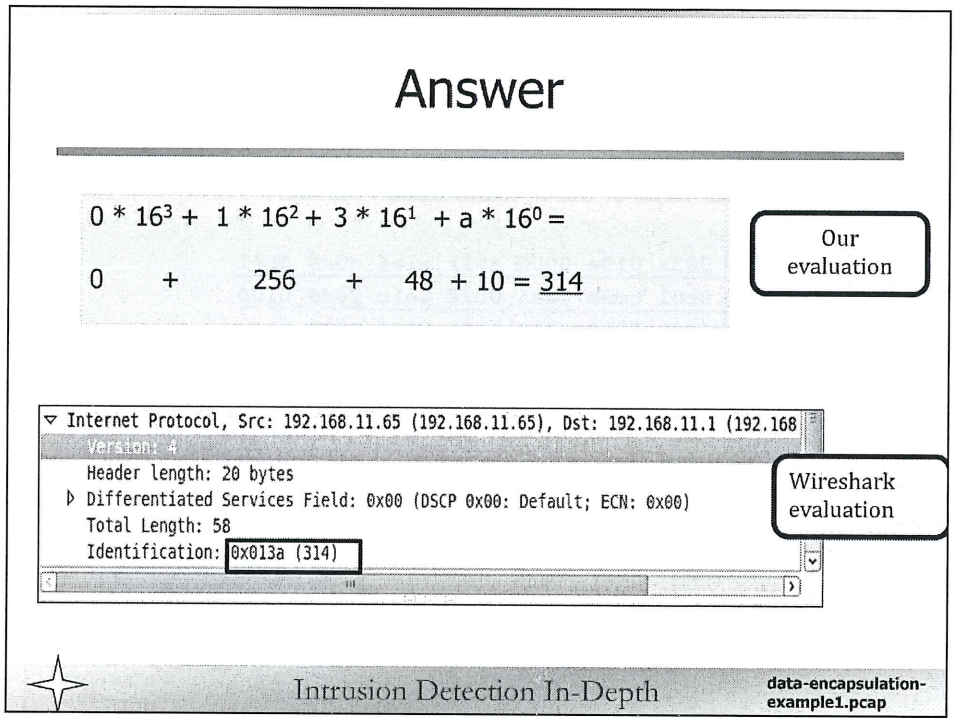
tcpdump -r data-encapsulation-example1.pcap -n -x

Now comes the hex output. There are 16 bytes (32 hex characters) per line. The line begins with the offset into the packet expressed in hex (like you really want more opportunity to do hex to decimal conversions). Remember when dealing with byte offsets into the packet counting begins at offset 0, **not** offset 1. Tcpdump makes no attempt to delineate the layers in the hex output for easier comprehension. The IP layer is in black text with a single underline. The transport layer, in this case UDP, has a ridged underline. Last is the application layer, DNS, with a dashed underline.



Being able to decipher hex values in an entire packet may come in handy. Well, at least with high certainty, it will be handy to know if you take the certification exam since it is considered a vital skill to have. Let's start analyzing packet data with this challenge. What is the decimal value associated with the IP Identification field?

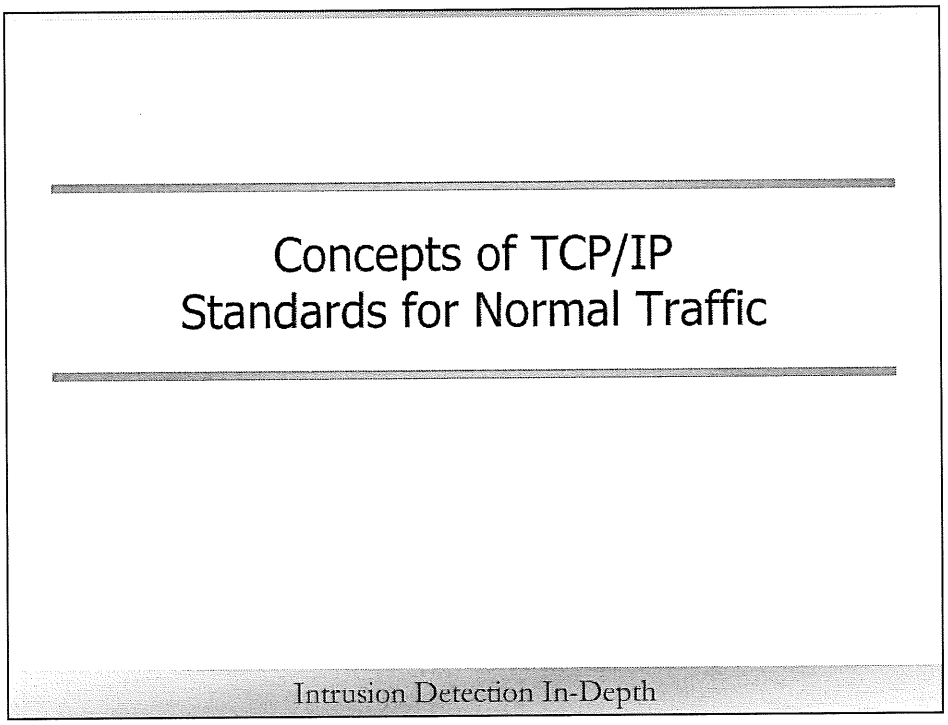
Remember the prescribed method of determining this. First, use your IP header reference to discover where in the IP header the identification field is found and how many bytes it is. Now, begin at the rightmost hex character of the field and represent each hex character from right to left as incrementing powers of 16. Once you have done that, multiply the base value by the exponent value and add all the individual results up to yield the answer.



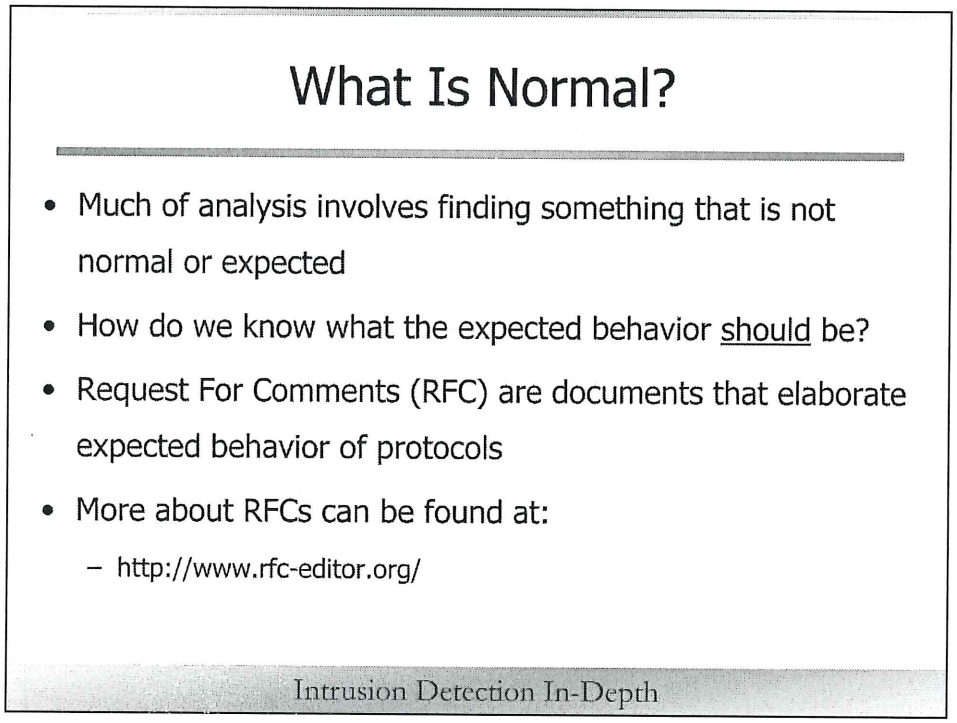
Here is an IPv4 header with the first 6 bytes of hexadecimal data found on the previous slide. If you take each hex character and place it in a nibble, you find that the value of the identification field is 0x013a. Once again, represent the values as powers of 16 beginning at the rightmost nibble of the field, in this case 0xa. This yields a value of 0xa * 16^0 that equals 10. The next hex character to the left is 3, and 3 * 16^1 = 48. The next hex character to the left is 1 and 1 * 16^2 = 256. Therefore the identification value in this packet is 10 + 48 + 256 = 314.

We can do a final check to compare our result with Wireshark's evaluation of the Identification field. As you see, it has a value of 314.

That does it for the fundamentals of bits, nibbles, bytes and translating from binary to decimal to hexadecimal and back! You should be more comfortable with hexadecimal dump output so that you can perform magical conversions yourself.



This page intentionally left blank.



Before we go off and start our analysis, we have to understand what is normal or expected. We need to know this because our job as an analyst often involves finding something that is unexpected or abnormal. But, what is normal? We can discover what is normal and expected about protocols by looking at documents called Request for Comments (RFC). A given RFC elaborates the expected standards for a particular protocol.

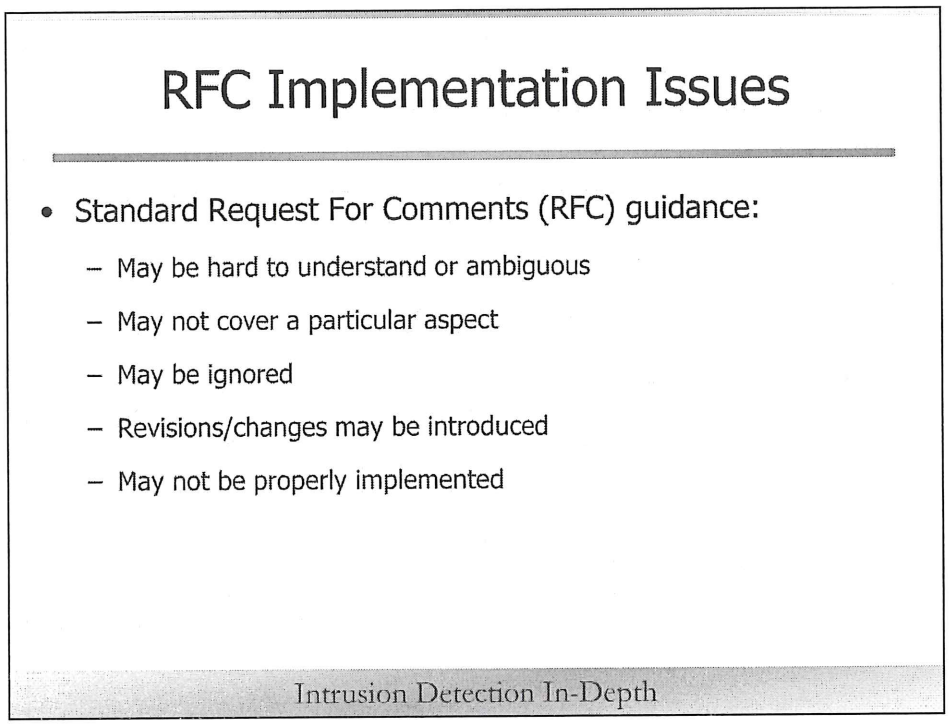
Once issued, RFCs do not change. Protocol revisions are documented by issuing new/superseding RFCs. RFCs of special interest may be:

RFC 793 - Transmission Control Protocol describes the functions to be performed by TCP, the program that implements it and its interface to programs or users that require its services.

RFC 768 - User Datagram Protocol which describes the functioning of UDP, which is an unreliable connectionless protocol.

RFC 791 - Internet Protocol or IP which discusses the protocol that provides for transmitting blocks of data called packets from sources to destinations.

RFC 792 - Internet Control Message Protocol (ICMP) which discusses the protocol to deal with an error in packet processing.



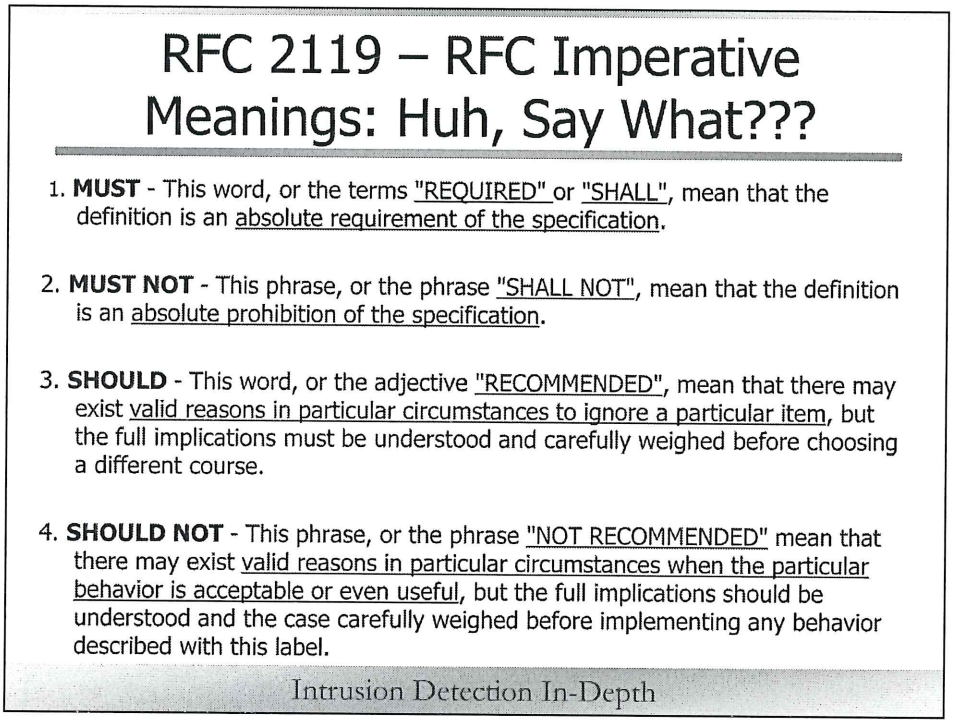
While guidance is offered by a given RFC, that doesn't necessarily mean that it is implemented properly or even implemented at all! If you've ever had the pleasure of reading an RFC for amusement or relaxation, you know that many of them are notoriously hard to understand – bordering on incoherent at times. Given this type of direction, it is easy to see why different implementations may vary.

In addition to being hard to understand or ambiguous, a particular RFC may not cover all aspects of a given protocol. For instance, what if TCP segments arrive that overlap each other? Should the first or subsequent one be honored? Also, some implementers disregard the RFC standards altogether and create their own incarnation of a given protocol. Microsoft has been known to do this a time or two. And, sometimes RFC's are revised and an implementer may not make the changes right away, if at all. Finally, there are those implementations that are just not done correctly.

As an IDS/IPS analyst should you care about RFC implementations? Let's take the example of the overlapping TCP segments. Let's say that the first segment contains some innocuous content and the overlapping segment contains a payload of the same length, but with exploit content. Further, let's suppose that the IDS/IPS honors the first segment and doesn't alert or block the exploit segment. Finally, let's say that the target host rejects the first segment and accepts the overlapping segment. If the exploit is successful, it goes undetected.

In this case, the exploited host's operating system TCP implementation elected to accept the overlapping segment. But, other operating system TCP implementations may accept the first segment. The only way for an IDS/IPS to deal with this and other ambiguities is for the IDS/IPS to have foreknowledge of the target host's operating system and to be aware of its TCP reassembly preference for overlapping segments. This is known as target-based awareness and is available in Snort versions 2.8 and later that employ the stream5 preprocessor. We'll discuss this concept in more detail later in the course.

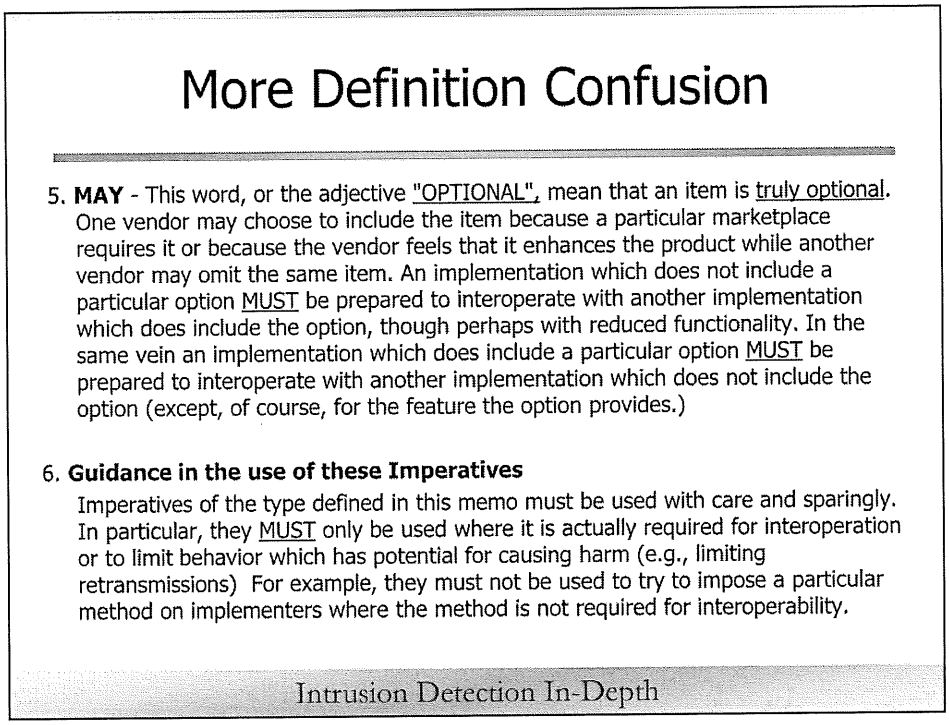
Exaple Snort knows how each OS aseamble fragment



RFC 2119 "Keywords for use in RFCs to Indicate Requirement Levels" defines words that describe the "imperatives" for a given facet of protocol implementation. There are five different definitions – two follow on the next slide. There are positive and negative conditions for all but the last requirement level.

I don't know about you, but I need a non-RFC to interpret this RFC☺! First, there are alternate descriptions of many of the requirements levels. "MUST", "REQUIRED", and "SHALL" are interchangeable. "SHOULD" is the same as "RECOMMENDED". For goodness sake – pick a single description for each and stick to it to eliminate confusion.

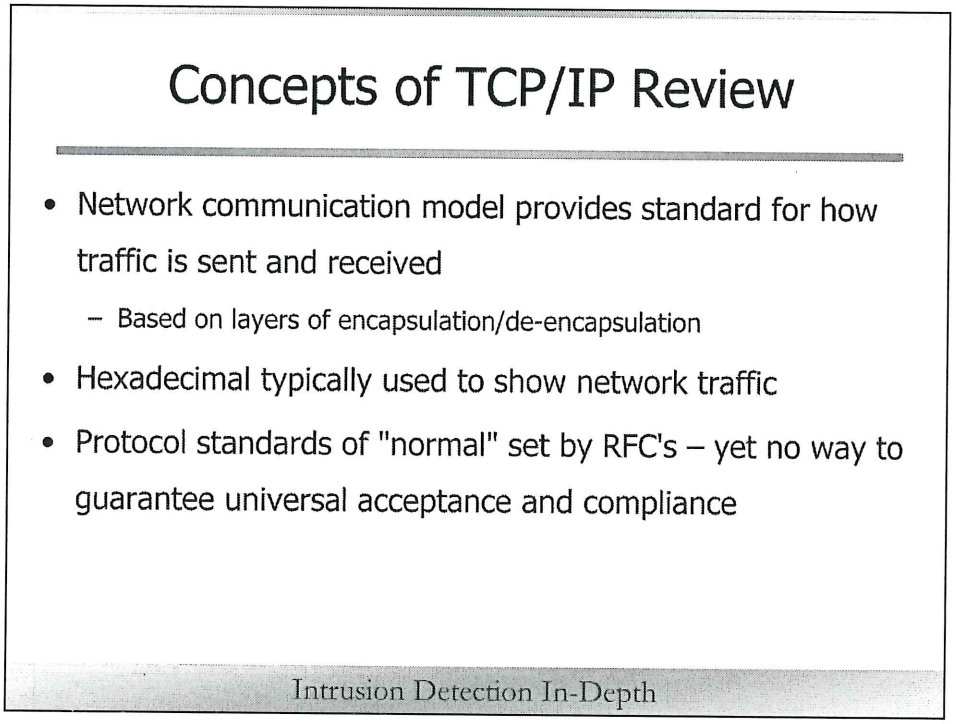
The first two imperatives of "MUST" or "MUST NOT" appear straightforward, indicating that implementation is or is not required. "SHOULD" or "SHOULD NOT" are less precise seeming to encourage or discourage implementation, but ultimately making it an implementer choice.



"MAY" is synonymous with "OPTIONAL" and appears to be "truly optional". But, if you read on you see conditions, expressed with the "MUST" verbiage to denote that inclusion or omission of a given feature must work with another implementation (perhaps another OS) of the same protocol feature that excludes or includes the given feature. If the text in the parentheses of the discussion of "(except, of course, for the feature the option provides)" leaves you shaking your head in befuddlement, you are not alone!

As if imperatives 1-5 are not baffling enough, the guidance offered for the use of these imperatives in section 6 takes the whole blurry jumble to a new level of confusion. It appears to be exhorting the use of these imperatives when creating the RFC only when advising about characteristics required for protocol interoperability or when possible harm may result from the lack of specific imperative use.

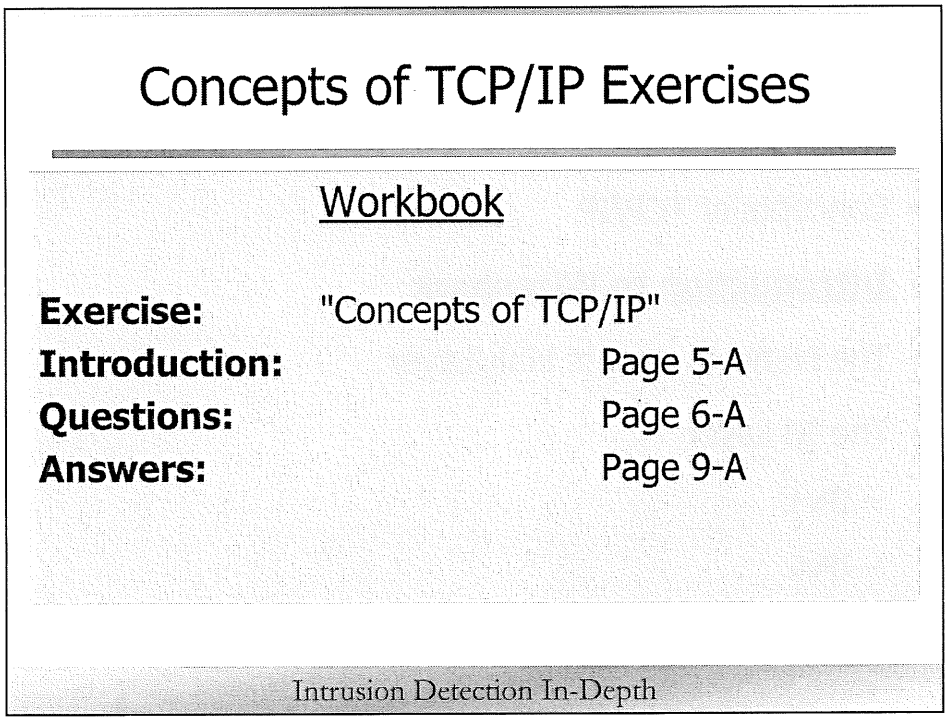
It is no wonder implementers interpret RFC's diversely. The particular RFC description itself may introduce unclear guidance. As well, the definition of the selected imperative for whether or not a given feature should be included is so arcane. The point is that determining what is "normal" behavior for a particular protocol implementation may be extremely difficult for the implementer to understand.



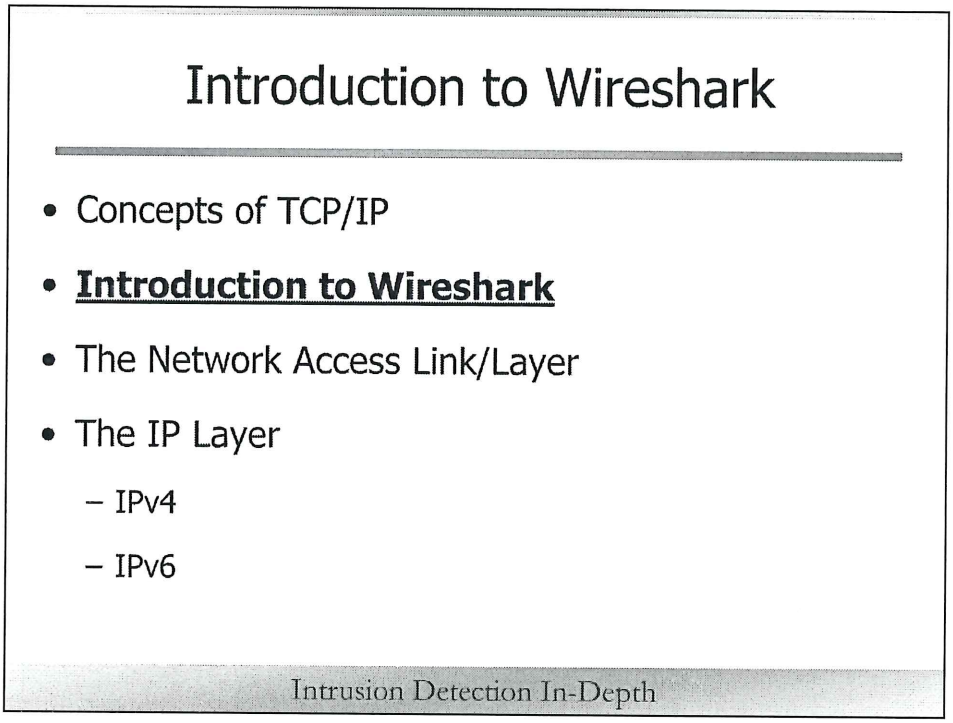
As we've seen the TCP/IP communication model provides a standard for sending and receiving traffic. It uses the notion of layers – specifically the network access or link layer at the bottom, followed by the IP layer, followed by the transport layer, and finally the application layer. Each layer encapsulates the next higher layer when data is sent. Each layer strips off the current header, and passes the remaining data to the appropriate higher layer, a process known as de-encapsulation.

We covered the topic of representation of data in binary and hex. Hex is typically used to show network traffic by tools like Wireshark. Data is represented in bits, nibbles (4 bits), and bytes (2 nibbles or 8 bits).

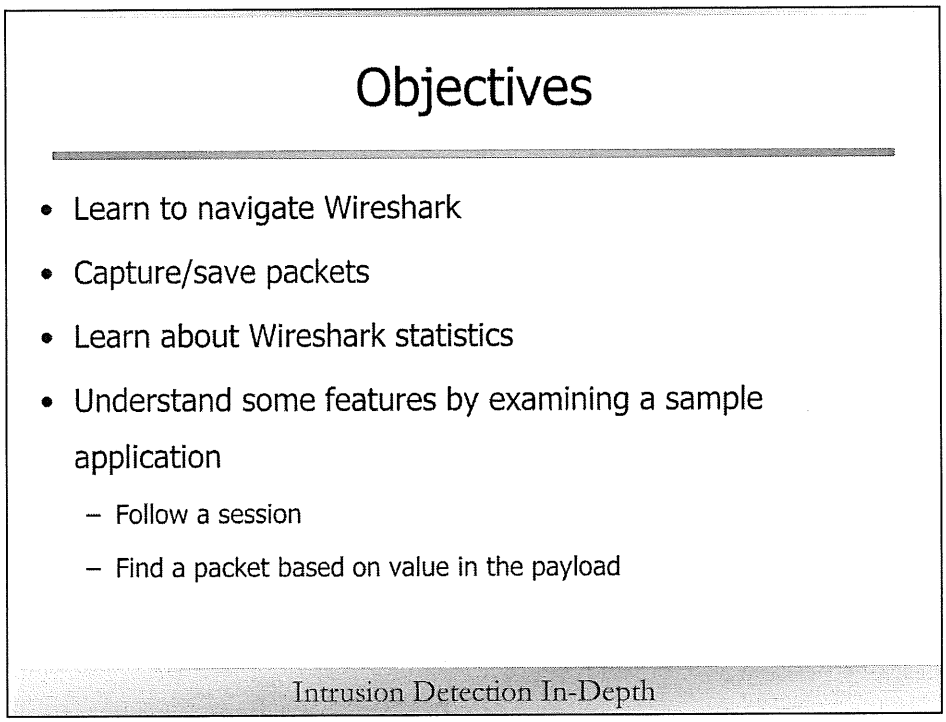
When a new protocol is created or an existing one amended, the Request for Comments documents act as guidance to suggest how to implement it. Yet, implementers may not follow the guidance for many different reasons. We hope to establish a foundation of normal using the RFC guidance so we know abnormal when we see it.



This page intentionally left blank.

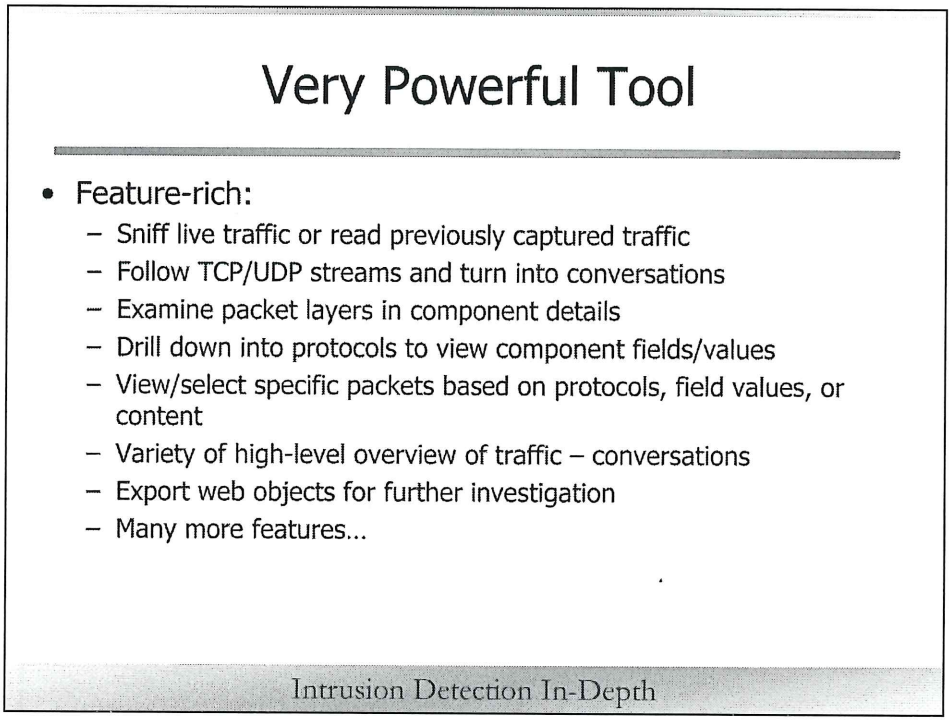


This page intentionally left blank.



This section shows you how to get around in the Wireshark interface. Wireshark has the capability to give you some statistics that are helpful to get an idea or overview about a particular pcap or set of traffic that you've captured.

The best way to demonstrate some of the features of Wireshark is to show you a simple application where we follow an entire session, allowing Wireshark to reconstruct many different related packets. We'll also see how a particular packet(s) can be found based on some value in the payload.

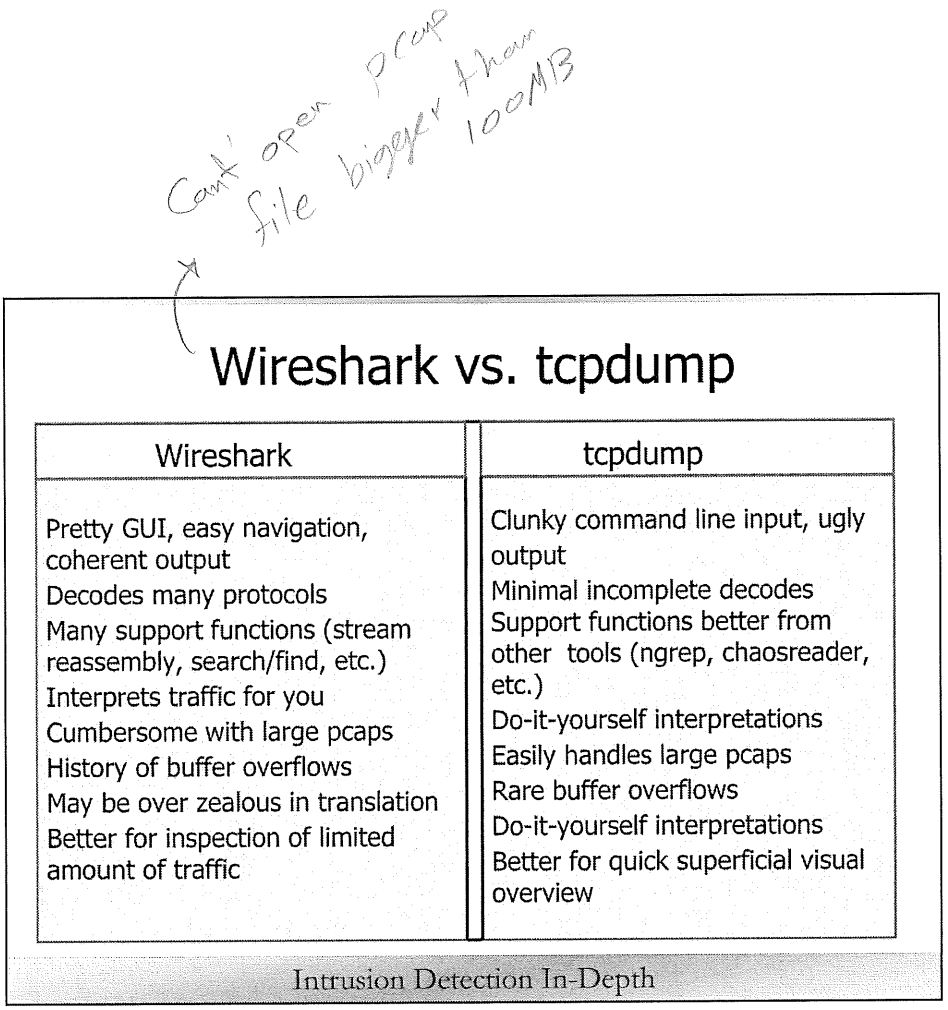


You had an opportunity to take Wireshark for a short test drive in the first section if you followed the Wireshark output on the VM. We'll learn to use some of the basic and necessary functionality in this section of Wireshark. Additional features and functions are covered in sections of following days' material. Wireshark has built-in features to help you examine any portion of any packet or packet content that you'd like. It can also decode many protocols into relevant fields and values. Wireshark allows you to search through records for particular traits. And, much like tcpdump, it can capture or sniff traffic and it can also read pcap files.

Another extremely useful feature is reassembling a TCP or UDP conversation. Wireshark is able to reassemble all segments in a TCP or UDP session to recreate each side of the conversation. This is very powerful for analyzing a particular session of interest. Wireshark is particularly adept at decoding many different protocols. This permits you to see values associated with specific protocol fields and allows you to search these fields for values; for instance, if you wanted to select all records that contain an "HTTP" GET method. This is possible because Wireshark can dissect the HTTP protocol into its component fields and values.

Wireshark offers many different configuration options. It also has many different views of the traffic, including a variety of overviews such as the source and destination IP addresses of all conversations and statistics of packets. It can also write captured records in many different formats and can export web objects.

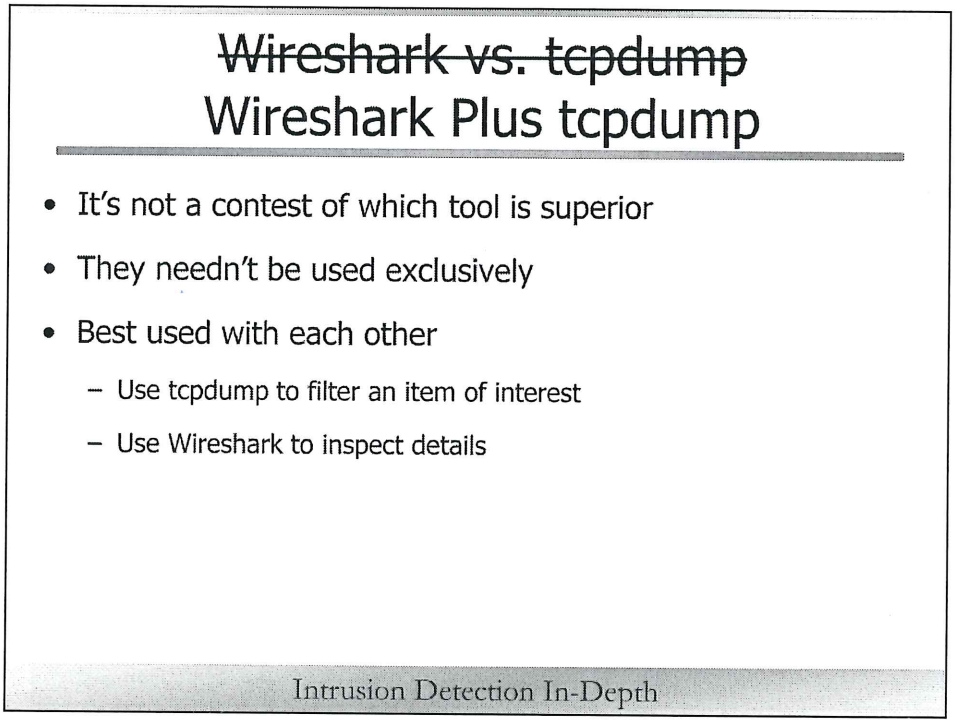
There are many Wireshark features that will not be covered in the course just because Wireshark has so many capabilities that all of them cannot be covered in the time we have. The references page at the end of this section points you to other resources in case you'd like to pursue learning Wireshark in more depth.



Both Wireshark and tcpdump will be used throughout the remainder of this course for packet display and processing. Many years ago, tcpdump was pretty much the only tool for packet analysis. It had been around for a long time when Wireshark (initially named Ethereal) came along. We'll spend a great deal of time using both tools and discovering their benefits and limitations. If you are new at traffic analysis and try both tools, you may think why ever use down-and-dirty tcpdump? It's got ugly output generated by esoteric command line directives compared to Wireshark's lovely GUI, menus galore, and coherent output.

Wireshark has support to decode many different protocols – some quite familiar – such as TCP, DNS, etc. Other protocols are more arcane, yet common and uncommon protocols are presented in a manner where you can see all the fields in the protocol and the associated data decoded for you. In contrast, tcpdump has minimal protocol decodes – such as DNS, and even so, the decodes are incomplete. You may see the DNS query that was requested, but you won't see other related fields and values of the query. You'll also find that Wireshark is what the marketing people call "feature-rich" or chock full of easy to use functionality for all sorts of tasks such as session reassembly, finding packet content, saving output in many different formats, and searching for any field in any protocol as a few examples. Tcpdump, on the other hand, has very few support functions and relies on other tools that we'll examine later in the course, such as ngrep and chaosreader for search and session reconstruction.

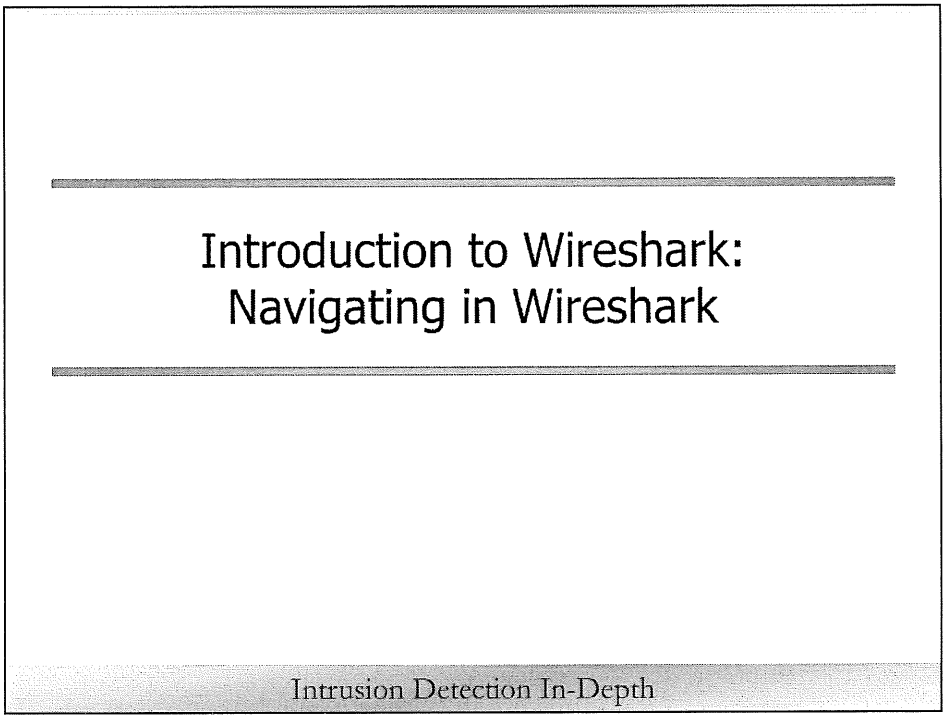
Basically, Wireshark interprets the traffic for you, packet by packet, protocol by protocol, and field by field. Tcpdump mostly requires you to do the interpretation. You may be thinking to yourself once again, why would I ever use tcpdump? Well, what if Wireshark code is wrong and misinterprets some kind of translation? Tcpdump can be used in its raw hexadecimal format to allow you interpret the protocol yourself. Also, if you try to bring up large pcaps – over 1GB or so, Wireshark groans and wheezes before finally coming up. Tcpdump can process large pcaps much better and far more quickly. Suppose you want to find a specific packet in a large or even medium sized pcap when you have a unique characteristic about the packet to supply as the search criteria. Tcpdump is quick, has minimal overhead, and is more efficient.



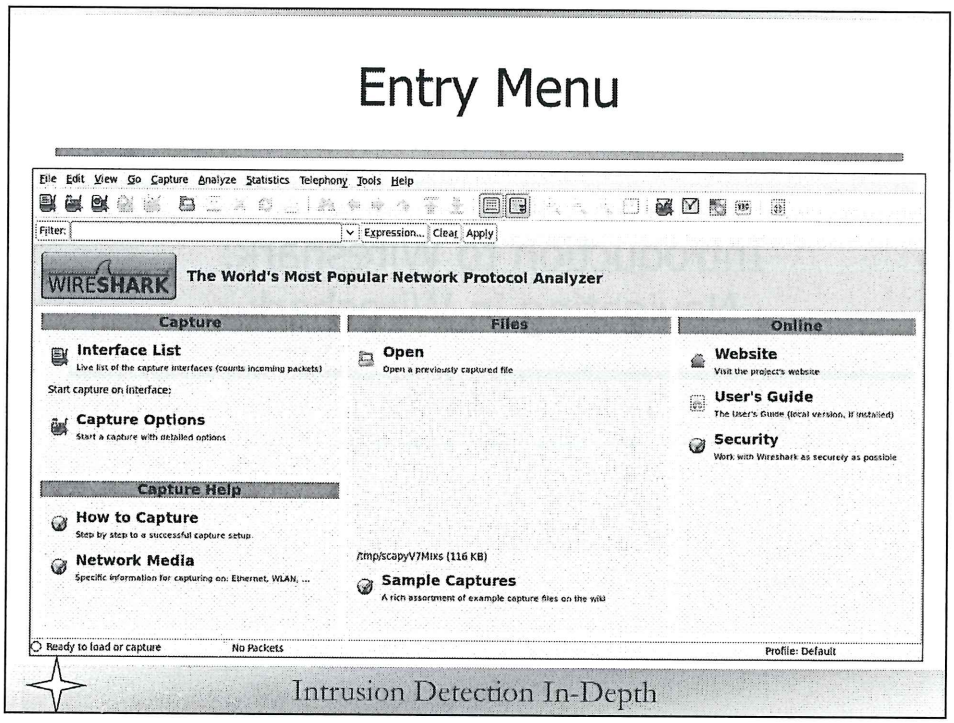
The comparison of Wireshark versus tcpdump was not meant to incite a holy war like Firefox vs. Internet Explorer versus Safari over which is superior. The reality is that they don't have to be mutually exclusively employed. There will be times when one or the other may suit your needs entirely. There may be other times when you will find that they can be used to complement each other.

For instance, say you have a very large pcap that tcpdump easily handles but that Wireshark is slow to ingest. Imagine that there is some unique trait of interest that can be easily selected using tcpdump filters. We'll cover these in much detail, but for now, know that they are a means for selecting one or more packets based on some value(s) in the protocol headers or payload. You can extract the records of interest using tcpdump and then feed them into Wireshark to inspect the details, perhaps reconstruct a session, or export objects.

We'll examine many different tools in this course. Some are more focused on a specific task, such as attempting to determine an operating system associated with a given packet. Others are broad and full-featured such a Wireshark. No one tool is likely to meet all of your needs so use each for its strengths. Ultimately, you may find that many used in combination deliver the results you want.



Let's first examine Wireshark's main viewing interface. Wireshark has many features and its interface is quite intuitive. It's important to become acquainted with the menus so that you know how to access its features to navigate your way around.

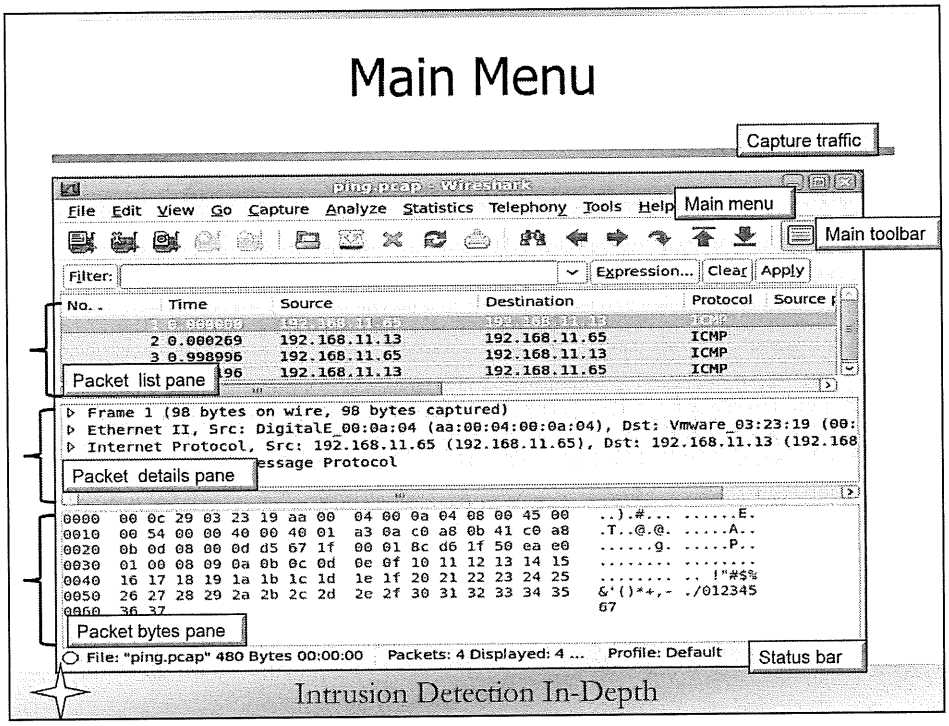


When you invoke Wireshark, this menu allows you to select options to configure the capture interfaces. And, it allows you to get help for capturing packets or access the user's guide online.

The middle column displays the pcap file names that were most recently viewed using Wireshark. The Wireshark website has gathered a collection of pcaps that can be accessed via the "Sample Captures" link. These are pcaps of some common protocols in case you have a particular interest in examining one or more of them.

Many of the toolbar items are grayed out since they pertain to functions that are used for a given pcap. For instance, the binocular icon finds a certain packet based on criteria that you supply. It makes no sense to have this function available unless there is a pcap loaded or traffic captured, which is not the case yet.

To invoke Wireshark from the command line, enter:
wireshark



This is the main menu for examining a particular pcap or the result of collecting live traffic. At the top, Wireshark's main menu options offer you choices for analysis using one of these pulldown options. The icons beneath the menu options comprise the main toolbar. If you hover your mouse over any of the highlighted icons, a brief description will appear.

There are three different Wireshark "panes". The packet list pane displays all the captured/read packets. This gives basic information about each packet. The packet details pane breaks down the selected packet by protocol. For instance, the selected record has a frame, Ethernet, IP and ICMP header. Each of these can be examined in more detail by expanding the protocol by selecting its associated right pointing triangle. The packet bytes pane shows the hexadecimal bytes and ASCII interpretation for the selected packet or portion of a packet.

The status bar on the bottom indicates the file name that has been read, the number of packets captured and/or displayed. The "Profile: Default" shows that the Wireshark configuration or preferences and settings it the default one that is supplied with Wireshark.

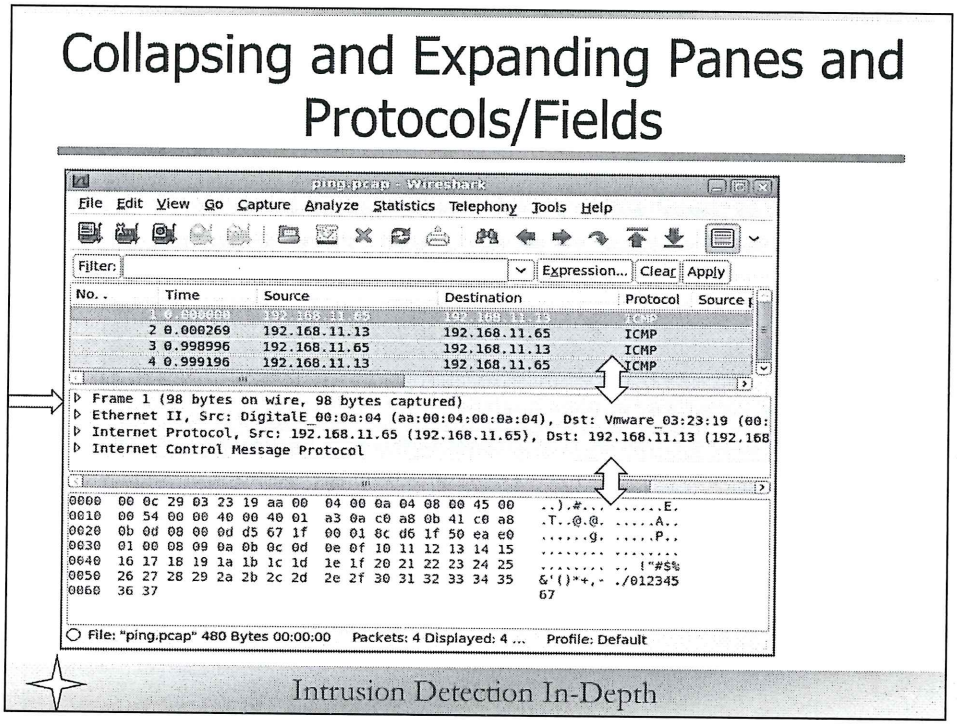
To display file **ping.pcap** do either of the following:

1. Exit Wireshark and invoke it again at the command line;

   wireshark ping.pcap

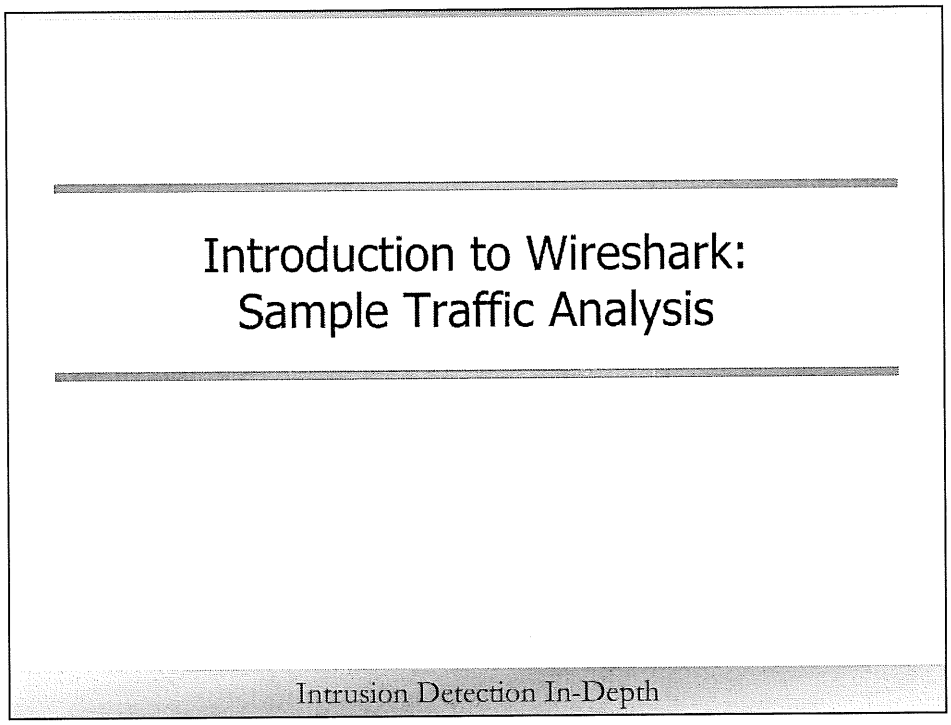
2. Or stay in Wireshark and select the pulldown menu File → Open and navigate to the directory location of **/home/sans/demo-pcaps/Day1-demos** and select **ping.pcap**

For viewing all future demonstration pcaps, you have this same choice, although the first option only is provided.

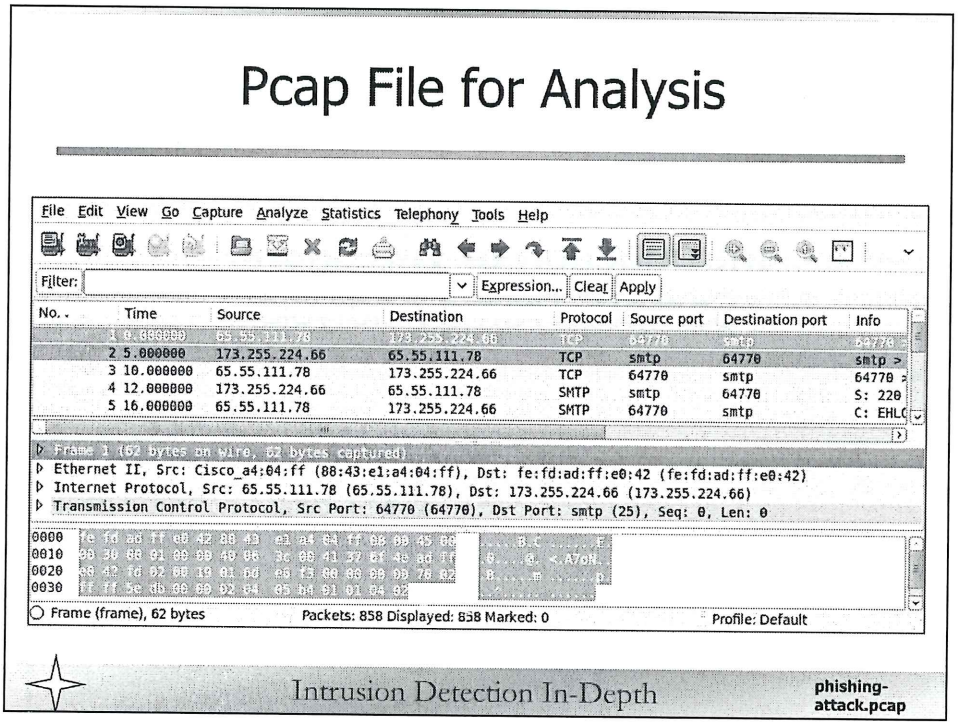


Wireshark presents you with a summary of the pcap data. You can expand and collapse display areas easily. Wherever you see a right facing triangle next to a protocol or field, for instance the "Frame 1" field where you see the arrow on the left side of the slide, you can expand the field by placing the cursor over it and clicking the mouse. Conversely, you can collapse the display by doing the same thing to a down facing triangle that replaces the right facing triangle when expanded.

You may also want to expand or collapse one or more of the packet pane areas to obtain a particular view of the data. There is a gray bar below the two left-right blue scroll bars separating the three panes that is used to expand and collapse the pane display area. Place your mouse on this scroll bar and a double ended pointing arrow appears. Just hold down the left mouse click and move the bar up or down to reveal or conceal a pane area.



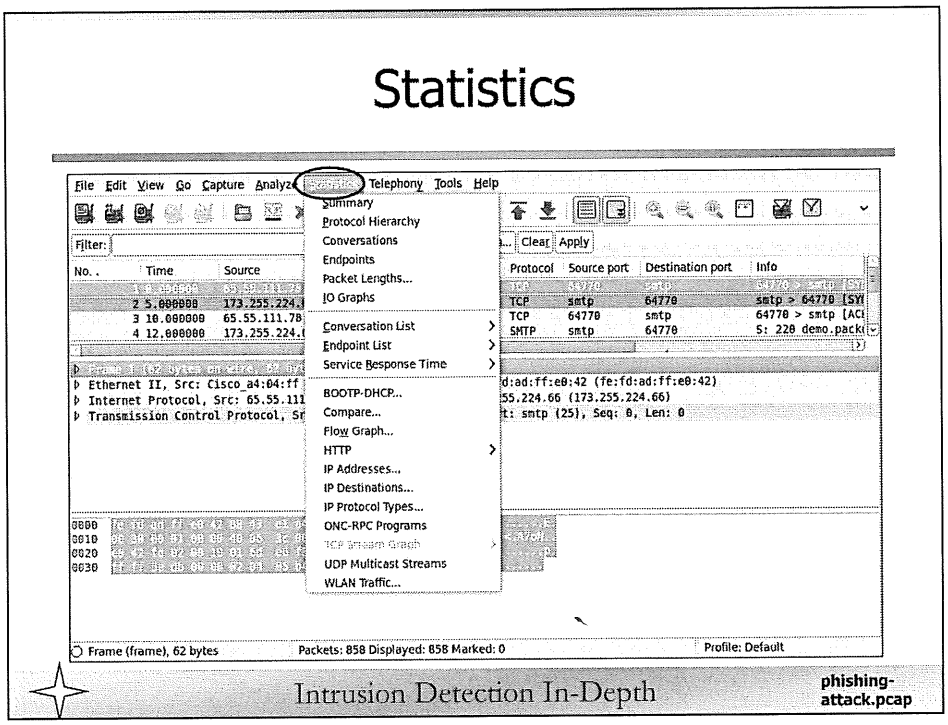
This section will introduce you to some of the more common Wireshark tasks you can perform when you are examining traffic.



Let's say that you have a pcap named phishing-attack.pcap, which as the name implies, is traffic of an attempted phishing attack. You are also given some more details that the string "filename=pdf641" in an HTTP session is indicative of a user visiting the link included in the phishing e-mail that directed the user to download some malicious code. That's all you know.

On the command line, open Wireshark to read the file "phishing-attack.pcap", located in the /home/sans/demo-pcaps/Day1-demos directory:

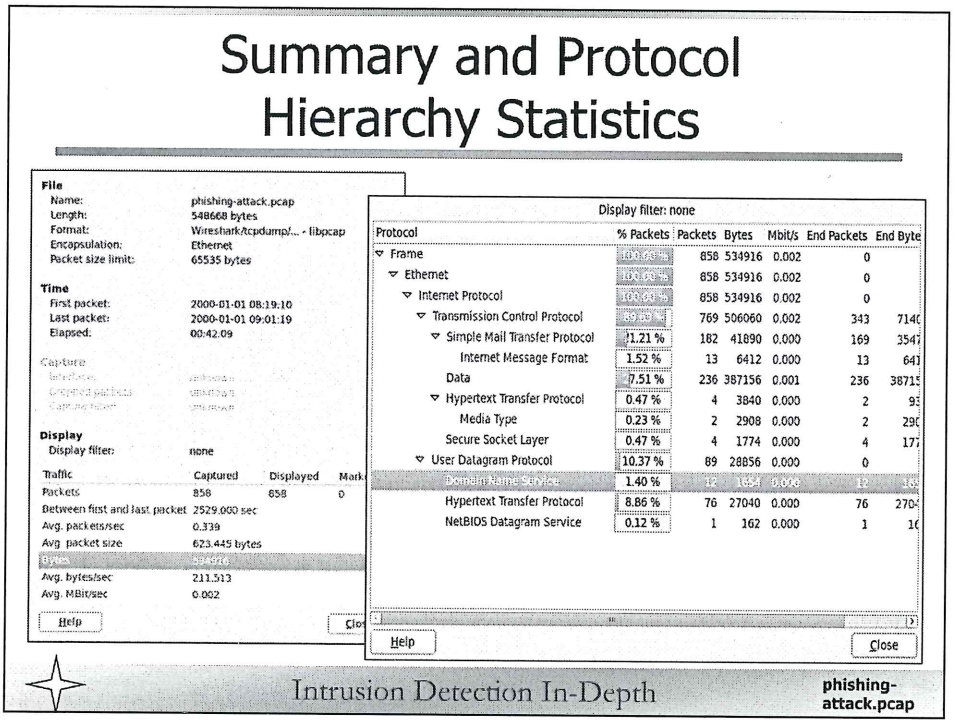
wireshark phishing-attack.pcap



Wireshark offers many different views and types of statistics associated with traffic. We don't know much about the composition of the records in the pcap – other than there must be some kind of Simple Mail Transfer Protocol packets and sessions and some HTTP session per the guidance given. Let's look at some statistics that may be helpful.

This is a great place to start when trying to profile some activity. It helps you get the big picture overview of what might be useful to investigate in more detail.

Select the pulldown menu "Statistics".

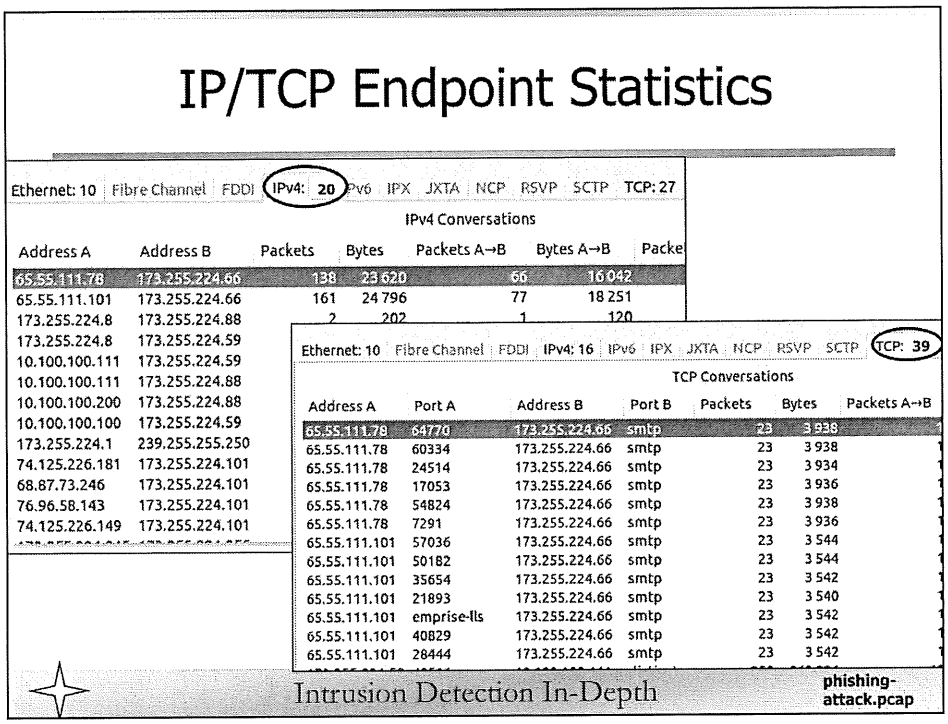


This output is the result of looking at two different options present in the Statistics pull-down menu. The display on the left is a Summary of traffic and the one on the right is the Protocol Hierarchy. The summary presents the number of bytes found in the pcap as well as the number of packets recorded. It displays the time when the first and last packets were captured. And, if you are interested, it gives a bunch of statistics of averages. You may notice that the file length under the File section at the top indicates that the file is 548668 bytes. Yet, when you look at the Display section at the bottom, the number of bytes is listed as 534916. The byte count represents the actual bytes in the packets only, whereas the file byte count includes those same bytes, yet has to keep metadata associated such as the timestamp for each record. The metadata is not considered to be packet bytes because it is not included in the packets themselves, only in the pcap file for statistics.

Another quite useful overview of the data in the pcap is found in the Protocol Hierarchy option in the Statistics pull-down menu. As you can see it categorizes the traffic by layers and indicates the approximate percentages, packets, bytes, etc. for each distinct category. This gives you an overview of the types of traffic that are found in the pcap as a good starting point for examining the data.

With your astute eye, you may see that there is over 8% of UDP HTTP traffic. What's that all about? As we'll later learn, if you click on the HTTP entry under UDP and right click, a filter menu appears. If you follow the Apply as Filter -> Selected options and click you will see the records that were included in HTTP UDP entry. The port 80 traffic is definitely TCP as that is the protocol found in the IP header of those records so that appears to be erroneous. The other traffic is identified as "SSDP". According to Wireshark's wiki: http://wiki.wireshark.org/SSDP, it is the Simple Service Discovery Protocol and it is described as an HTTP-like protocol. This explains why Wireshark considers it to be UDP HTTP traffic.

To view this output, select the Statistics pull-down menu. The output on the left is a result of selecting the Summary option while the output on the right is generated from selecting the Protocol Hierarchy option.



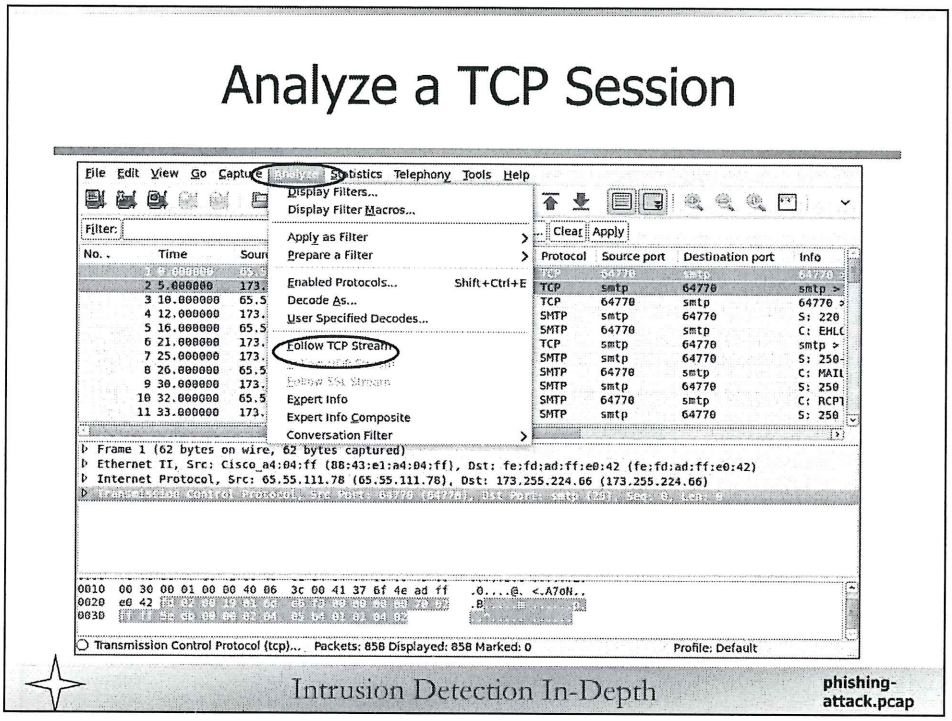
The Statistics option Endpoints provides good insight into the IP addresses and/or TCP/UDP ports. This is more helpful with smaller pcaps or larger more uniform ones with fewer IP addresses and protocols. The IP endpoints on the top of the screen, selected with the IPv4 tab, displays statistics for each unique IP of number of packets, bytes, transmitted and received packets and bytes. There are even some country designations based on IP address allocations.

Similarly, TCP endpoints below, selected by the TCP tab shows those same types of statistics by port associated with a given IP address.

You may notice the columns for country and city. Geolocation is available in Wireshark; however, it does not come natively. You need to download database and software support for geolocation. For more information on how to use it, look at the Wireshark wiki link:

http://wiki.wireshark.org/HowToUseGeoIP

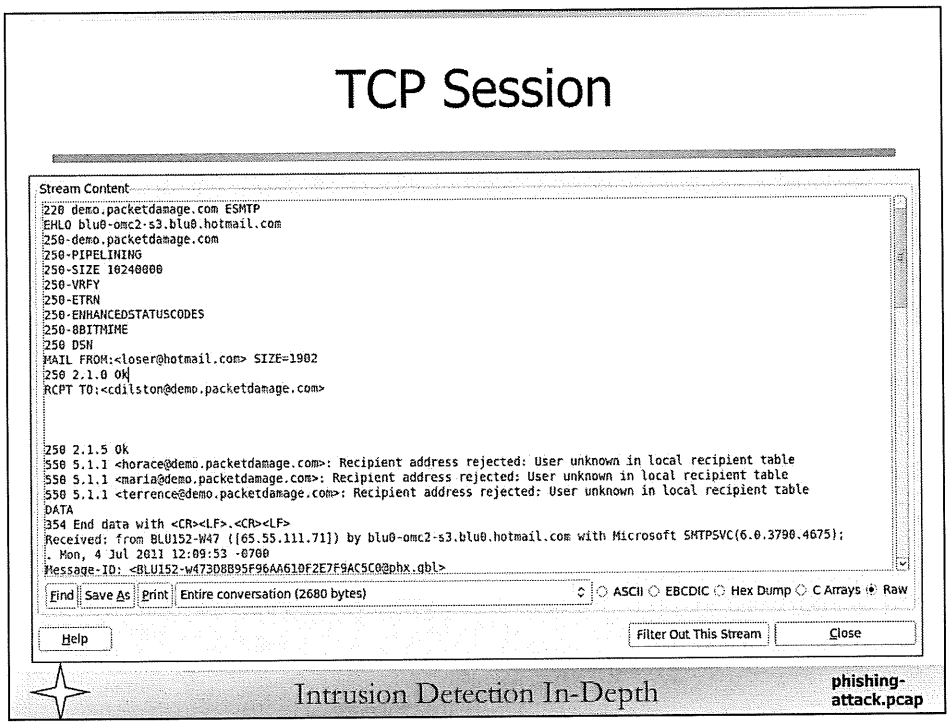
- Select the pull-down menu Statistics → Endpoints. Select the IPv4 tab to view the output in the upper display in this slide, and the TCP tab to view the output in the lower display.



One of the most power features of Wireshark is the capability to reconstruct TCP or UDP sessions. Tools like tcpdump are capable of seeing and analyzing traffic as single packets. That's fine if you are looking for something in an individual packet or two, but inadequate when you would like to see related packets in terms of a reconstructed stream or session. You are able to see both sides of the "conversation" in terms of the payload that is carried by all packets in the session. This is invaluable when trying to reconstruct what has transpired.

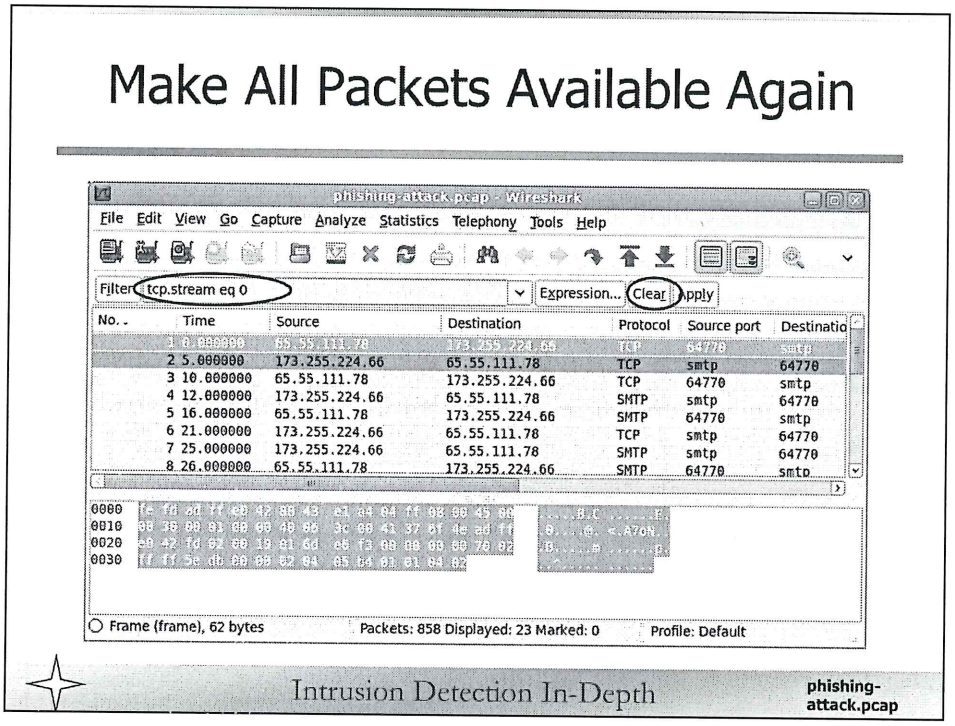
Let's follow a TCP stream in our phishing-attack.pcap – namely the session or stream associated with the first packet. You must select any packet associated with the session and then select the Analyze pulldown menu and the Follow TCP Stream option.

Select the first record displayed by Wireshark. Next, select Analyze → Follow TCP Stream option. You should see the output that appears in the next slide.



This is the actual "conversation" that occurred in the selected TCP session associated with the first packet in the pcap. You cannot tell from the black and white output in the slide above, but when examining this pcap in Wireshark, the client side of the conversation is in blue and the server side in red. This particular stream content is SMTP. We'll cover the SMTP protocol later in the course. But, this gives you a good idea of the power of Wireshark. Stream reconstruction is one of the most used and most valuable functions of Wireshark.

- Close the Follow TCP Stream window.

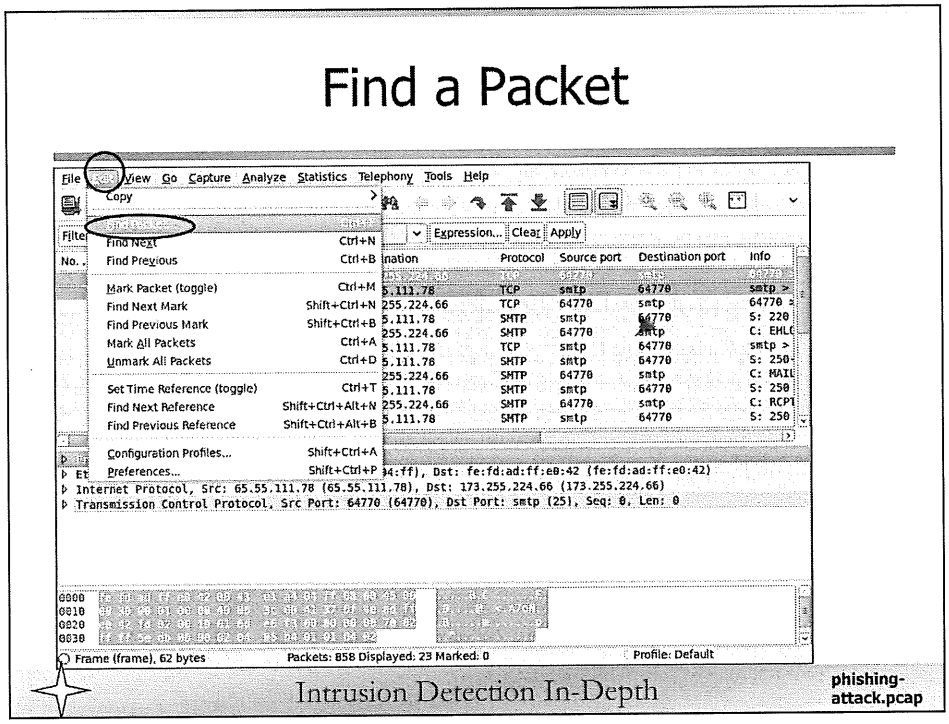


Whenever you select a function in Wireshark that selects a given session or selects packets with a specific trait, you exclude all other packets from Wireshark's present evaluation. Once you are finished looking at your selected packets, you need to inform Wireshark that you want to see all packets again.

You might wonder what operations cause Wireshark to exclude records. There are many. If you see the Filter text block with text and a green background, that means that Wireshark is able to perform tasks exclusively on those packets. Filters that you or Wireshark create in this text block are known as display filters. You will have an opportunity later to create your own display filters - as the name implies display certain packets only. In this instance, Wireshark is considering packets from "tcp.stream eq 0". Wireshark numbers its streams and the stream assigned the number 0 is the one currently active. You must clear the current filtered session to bring back all packets. Once you select Clear, the filter input becomes blank again and all records are once again available for evaluation.

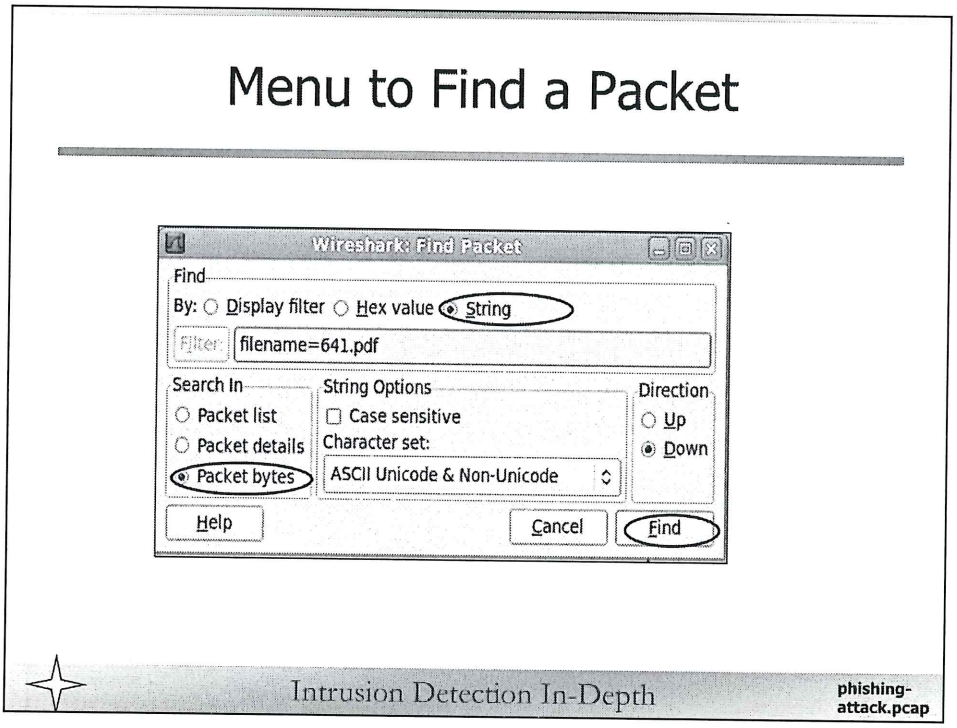
One distinction that you should understand is the difference between the results of Wireshark display filters and Wireshark capture filters. Wireshark display filters simply show those records that match, however all other records are still available after the "Clear". Wireshark capture filters select only those packets matching the filter criteria and reject /do not collect those that do not match.

- Select the clear button will make all packets available again.



Another very useful capability found in Wireshark is finding a given packet based on some indicator that you supply. Select the Edit→Find Packet menu option to bring up the menu to designate what you want Wireshark to see.

✦ Select the Edit→Find Packet menu.

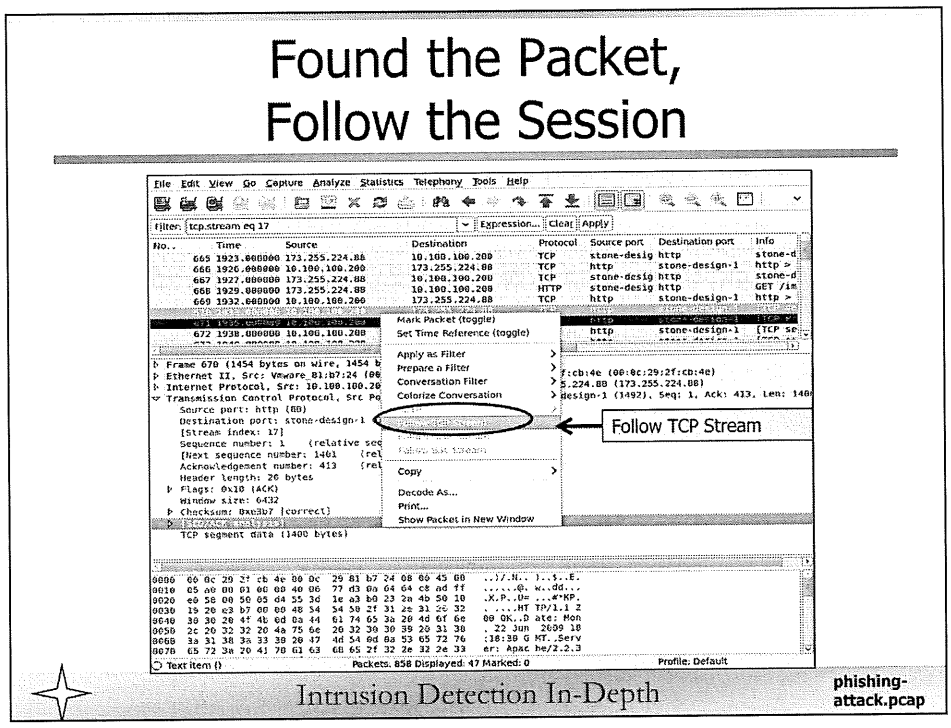


The dialog box seen in this slide is used to supply the details to Wireshark to find the packet you want. Remember earlier that one of the facts you received about this traffic was that string "filename=641.pdf" is likely to be associated with malicious traffic. Let's search for that. We are looking for a string value and we want to look in the packet bytes – or payload for the string.

There are other search options. You can search for a hexadecimal value in either the packet details (header fields) or packet bytes field (payload). The display filter option allows you to find the packet based on display filters that Wireshark supports. We'll go into great detail about display filters, but for now understand that Wireshark has its own designations for protocols or fields in protocols. For instance a Wireshark display filter of "udp.port = = 53" informs Wireshark to examine UDP port 53 (typically DNS) traffic only. The display filter selection does not allow you to select a "Search in" button – all of them turn gray. That's because this is a self-contained search and Wireshark knows where to apply its own display filters.

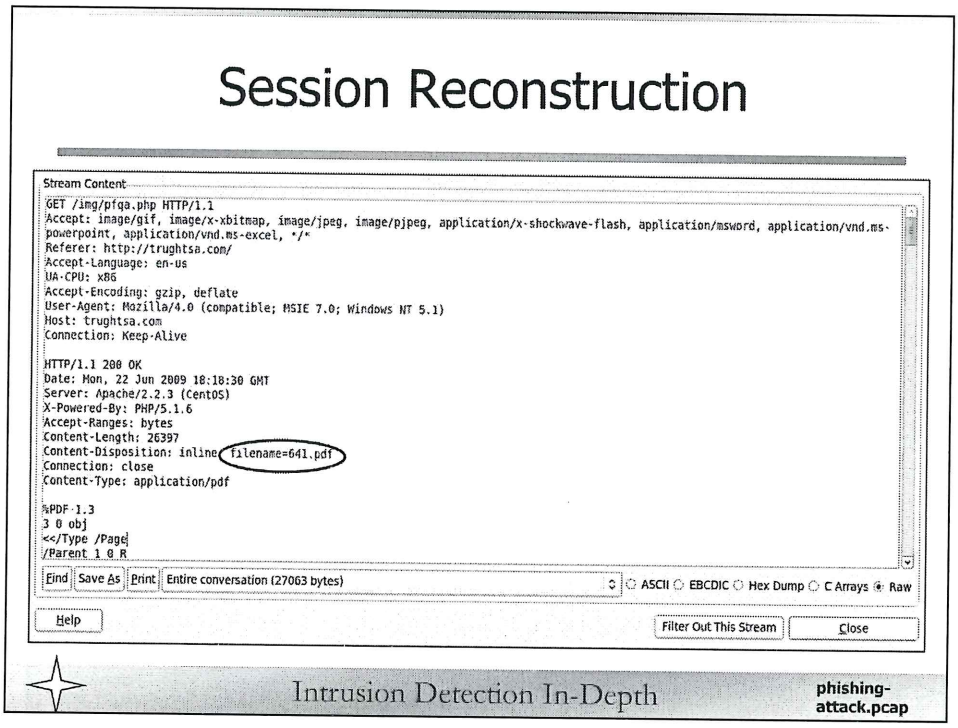
The "Search in" section has a selection of "Packet list" which means that you must supply a string format for Wireshark to match with what it calls the Info column. This is the column that has headers like Source, Destination, etc. By the way, just hovering over the "Packet List", or any of the other choices in this dialog box causes Wireshark to give you a short description of the field. Speaking of help, Wireshark has extensive help support under the Help pull-down menu option.

- Select the "String" option, enter a filter of "filename=641.pdf", select the Packet bytes option and hit the Find button. If you did not press the Clear button to restore all records after selecting Follow TCP Stream, you will not find the string.



Wireshark takes us to the packet where the text is found. But, it is more helpful if we can see the string in the session related to the found packet so we'll combine this with the Follow TCP Stream capability. Wireshark has navigated to the packet so all you have to do is right click the mouse and a menu is presented that has Follow TCP Stream. Select that menu option. The result is shown in the next slide.

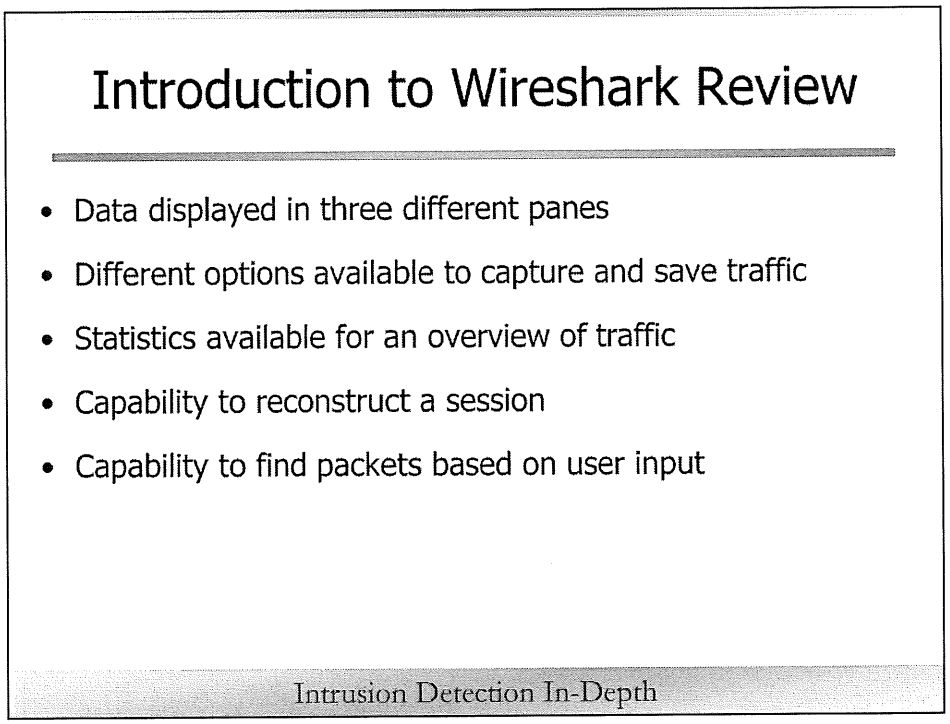
Wireshark has navigated to the packet so all you have to do is right click the mouse and a menu is presented that has Follow TCP Stream. Select that menu option.



You see the HTTP session where the user has requested a file. The HTTP server returns some information in the second section above (in blue if you are following along in Wireshark) including a header that has "filename=641.pdf". If we wanted, we could investigate the rest of the session reconstruction to see what transpired.

✦ Close the Stream Content stream.

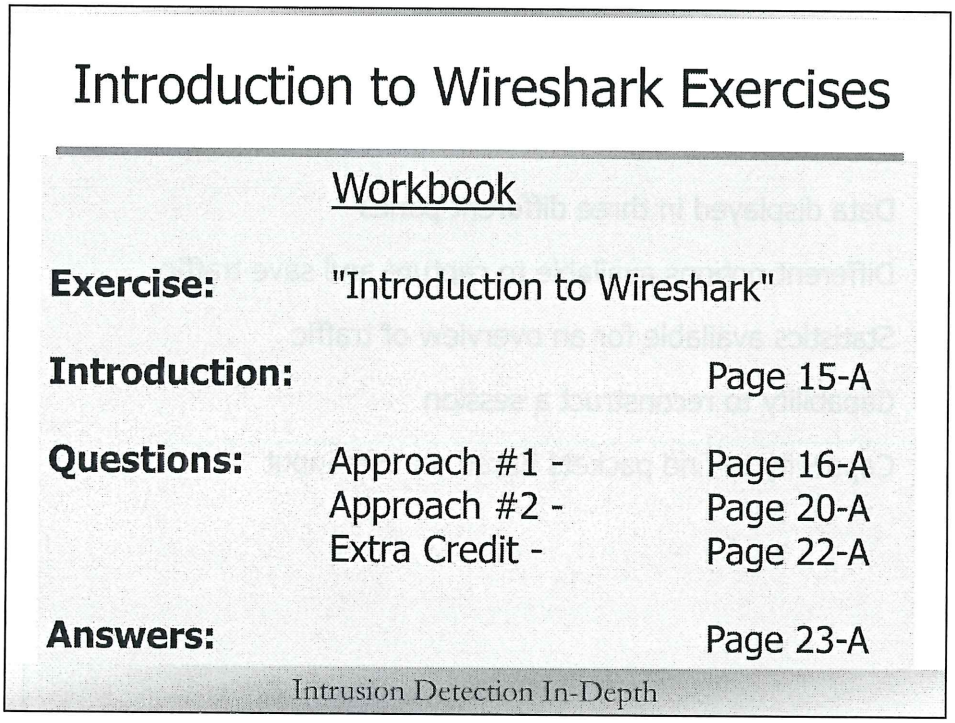
Exit Wireshark for now.



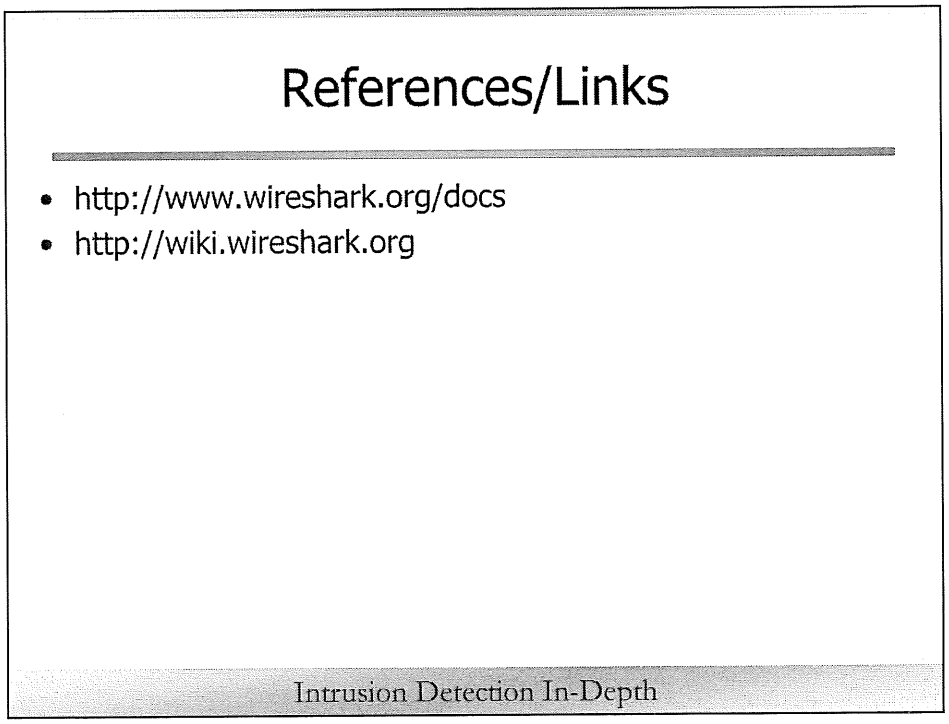
Wireshark has three different display panes – the packet list pane to show a one line summary of each packet, the packet details pane – one of the most useful features to dissect protocols and show you individual fields and values, and the packet bytes field that shows you the hex output.

Wireshark has a Statistics tab that displays many different types of overviews of the traffic, including conversation endpoints, protocols used, and a hierarchy of protocols all with summaries of the number of bytes and packets exchanged.

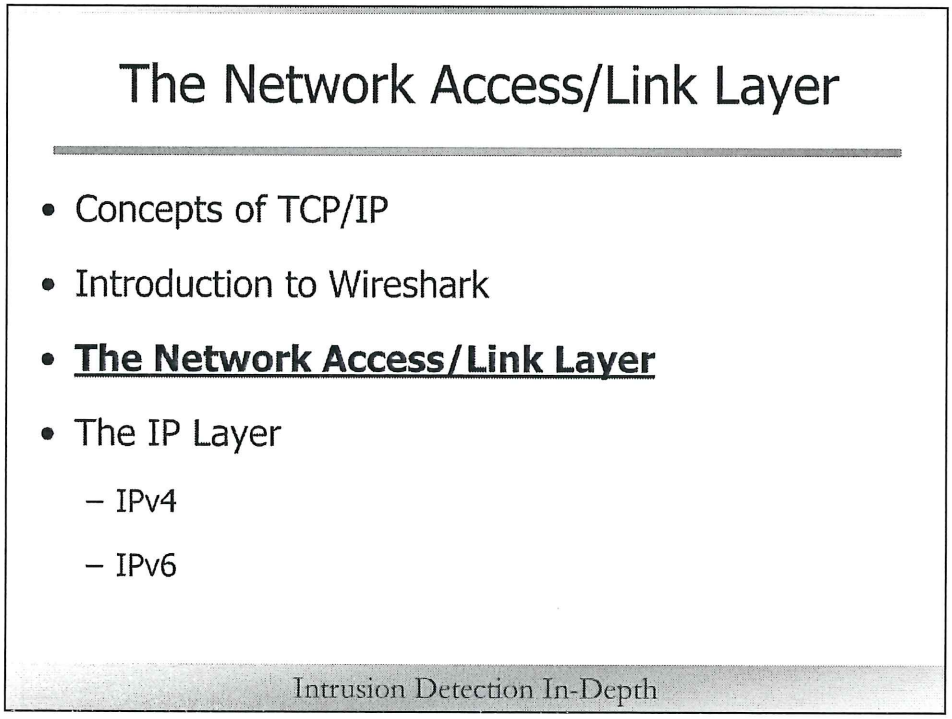
One of the most used Wireshark capabilities is to take multiple associated packets and reconstruct that stream of the entire conversation. This is invaluable when attempting to determine what transpired in a particular session. Finally, Wireshark is able to find one or more packets based on user input for a payload value, a protocol header value, or a particular Wireshark header characteristic.



For just about all of the exercises Approach #1 and Approach #2 contain the same questions. Approach #1 gives more guidance and hints to assist in answering the questions.



The information found at these Wireshark website links is very detailed and comprehensive.



This page intentionally left blank.

Objectives

- Introduce the 802.x link layer
- Discover how the network access/link and IP layers communicate using ARP
- Examine how ARP is not a secure protocol
  - How to spoof ARP
  - Consequences of ARP poisoning

The network access layer, also called the link layer, is based on the Institute of Electrical and Electronics Engineers (IEEE) 802.x standards. Our primary focus with the link layer is the need for a mechanism to communicate between link layer MAC addresses and IP layer IP addresses. The Address Resolution Protocol (ARP) protocol accomplishes this translation.

ARP is not sophisticated, subjecting it to misuse. This is accomplished with ARP spoofing resulting in ARP cache poisoning. This allows an attacker to launch a man-in-the-middle (MITM) attack.

IEEE 802.x Link Layer

- A family of standards developed by Institute of Electrical and Electronics Engineers (IEEE)
- Most link layers you will use fall in the 802.x family
  - 802.3: Ethernet
  - 802.11: Wireless
  - 802.15.1: Bluetooth

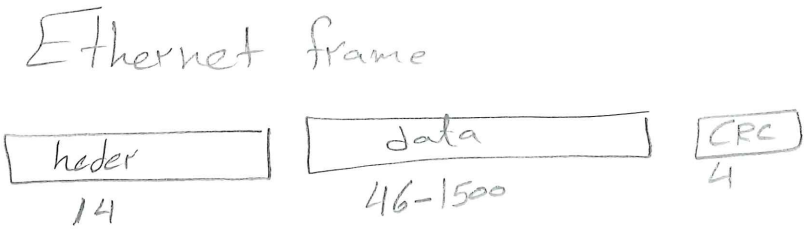
Intrusion Detection In-Depth

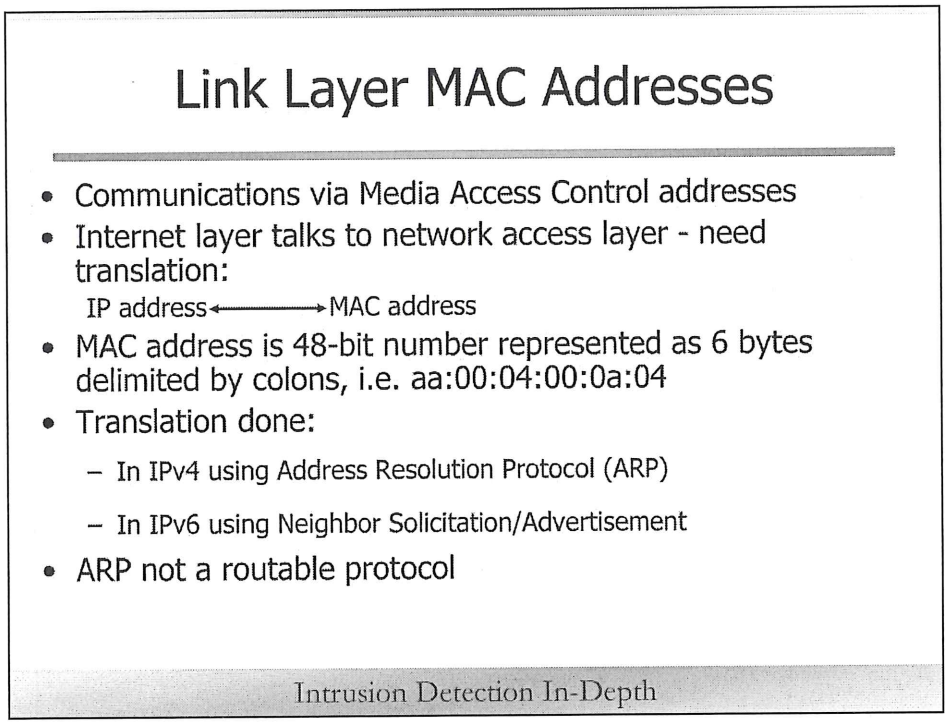
The 802.x link layers are a family of standards developed by a professional association known as the IEEE – usually pronounced I triple E. They maintain the standards for existing technologies and recommend standards for new technologies for the 802.x family. The name 802 is alleged to have been the year and month that the committee first met in February 1980. We'll examine 802.3 Ethernet traffic for the course, however you may be familiar with, and even capture, other 802.x protocols such as 802.11 – wireless, and 802.15.1 – Bluetooth.

The term link layer frame refers to the Ethernet header plus all the layers that follow. Ethernet supports a maximum size of 1500 bytes of data that follows the Ethernet header itself. The data that follows can be IP and a following transport protocol and possibly data or it can be ARP.

The Ethernet header itself is 14 bytes long. The minimum Ethernet frame size is 64 bytes. If there are less than 64 bytes from the combined 14 bytes of Ethernet frame header and data that follows, a trailer of 0's must be added to pad the number of bytes to 60.

The Ethernet frame has a 4-byte trailer also known as the CRC (Cyclic Redundancy Check) that is used to detect frame corruption. Thus, the maximum 802.3 Ethernet frame length is 1518 bytes of which 18 bytes are Ethernet protocol overhead. The remaining 1500 bytes can be used for user data such as an IP packet. The 4-byte trailer is *not* captured by tcpdump and most other sniffers since it is really not considered to be data.



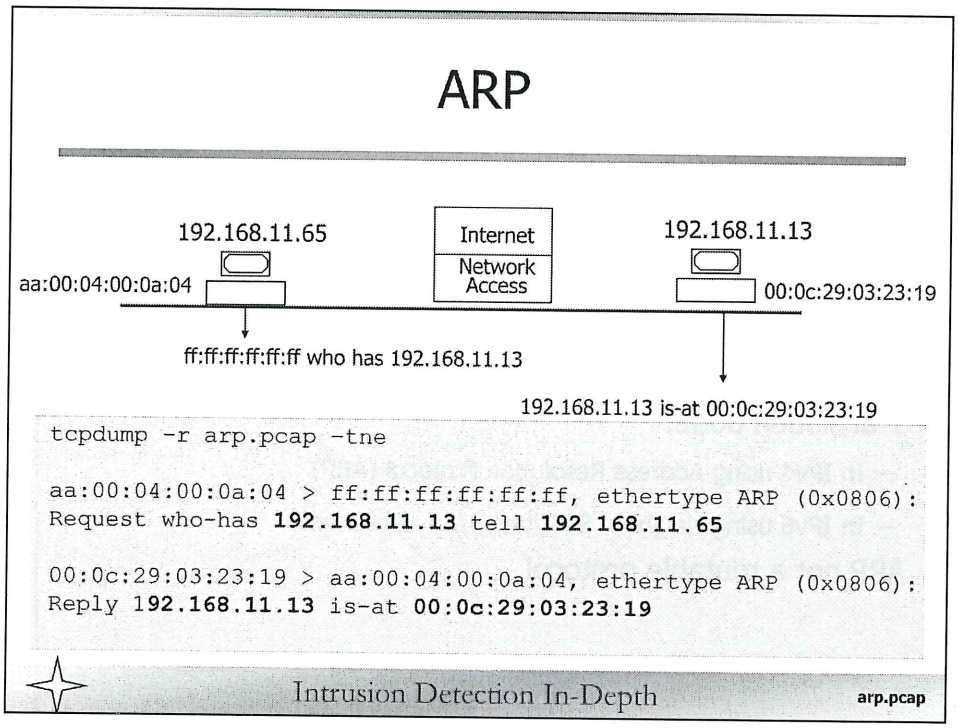


The network access layer involves the device drivers and the Network Interface Card (NIC) that must be used to communicate between the host and the physical medium on which it resides. A NIC has a MAC address burned into it. These are also known as hardware addresses that are 48-bit numbers that are not changed. The IPv4 address is a 32-bit software address that can be changed. Similarly, an IPv6 128-bit address can be modified.

When the network or IP layer talks to the link layer, an IP-to-MAC address association must occur. The Internet layer speaks in IP addresses and the link layer speaks in MAC addresses. ARP is responsible for making the association between an IP address and a MAC address in IPv4. IPv6 uses something known as a Neighbor Solicitation to issue the request for a MAC address associated with a given IPv6 address and a Neighbor Advertisement to send the response. We'll examine these transactions when we study IPv6. In IPv4, the host that needs to send the IP packet to the network access layer will issue an ARP request to determine the MAC address of the receiver. Once determined, a frame header will be constructed that contains both the sender's and the receiver's MAC addresses.

The first three bytes of a MAC address represent an Organizational Unique Identifier (OUI), designating the manufacturer of the card. For instance a MAC address that begins with 00:1F:33 indicates that Netgear produced the card. The last three bytes are the device ID.

One thing that you should understand is that MAC addresses are used for hosts found on the local network segment. If you need to send traffic to a host on another physical segment, it will have to be routed there using the Internet layer first before being delivered to the network access layer. What this means is that ARP is not a routable protocol since it resolves MAC addresses for local segments only. When a frame traverses a routing device, the existing source MAC address in the frame is substituted with the source MAC address of the routing device.



In this slide, we see ARP in action. We have host 192.168.11.65 that wants to talk to a host on the same local segment with IP address 192.168.11.13. Because both hosts reside on the local segment, we need to use the network access layer to send a frame from source to destination. Yet, 192.168.11.65 doesn't know what MAC address corresponds with IP 192.168.11.13.

So, it sends a broadcast ARP request to all MAC addresses on the local segment using a broadcast destination MAC address of ff:ff:ff:ff:ff:ff. The broadcast ARP request asks any host to respond if it has IP address 192.168.11.13. When the broadcast frame is read by the host with IP address 192.168.11.13, it will respond with a unicast (one host to another host) frame to the requester with the MAC address for 192.168.11.3 - 00:0c:29:03:23:19. The reply knows to return the response to the MAC address of 192.168.11.65, because it was included as the sending MAC address in the broadcast ARP request.

Now that 192.168.11.65 knows that the MAC address associated with 192.168.11.13 is 00:0c:29:03:23:19, it can supply it in the frame header and place it on the network access layer and the two hosts can communicate seamlessly. The IP layer that follows the frame/link layer header includes an IP header with the destination address of 192.168.11.13

Another important concept to introduce at this point is cache. The MAC addresses will not change legitimately unless a new NIC is placed on a host. So it can be stored on other hosts for a while to reduce the number of broadcasts required to support communications. Also, any hosts on the network that are listening for broadcasts will see the initial ARP request and can cache the requester's MAC and IP addresses for future communications with the requester. This too will reduce the potential number of ARP requests (broadcast traffic) that consume bandwidth.

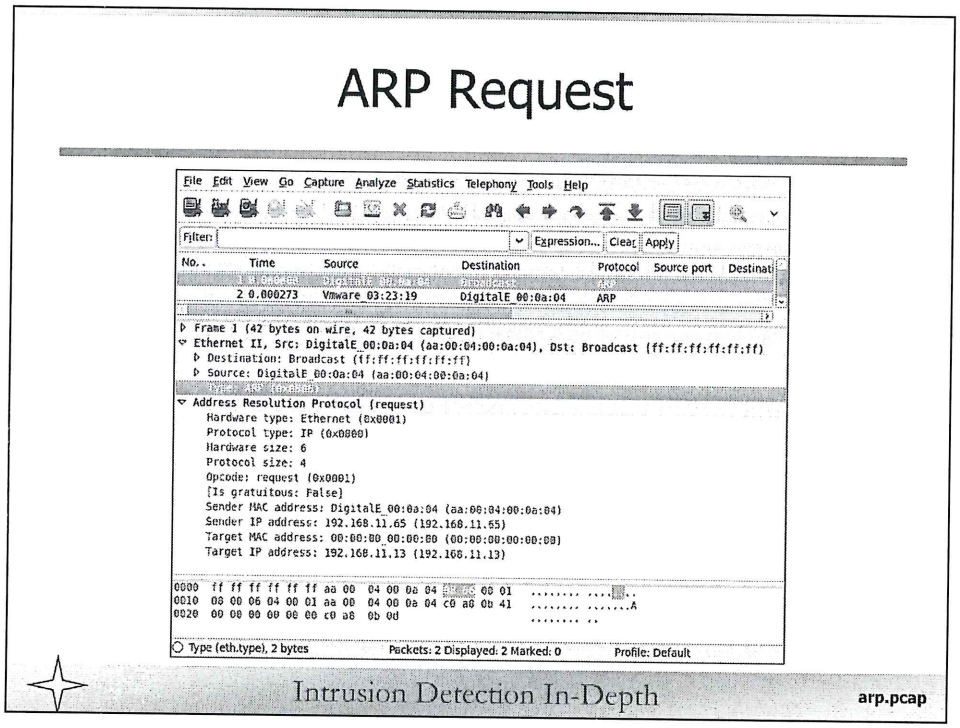
To see the traffic displayed enter the command:
tcpdump –net –r arp.pcap

This traffic was displayed using tcpdump is an abbreviated interpretation of the traffic. We'll see the Wireshark interpretation in the next two slides. Tcpdump uses command line switches to determine how to display the traffic and it uses filters – also known as Berkeley Packet Filters (BPF) to select which traffic to display. We'll cover the topic of BPF in much more detail in material later on in the course. This particular tcpdump command uses the command line switches of -n and -e and -t.

Command line switches can be stacked or piggybacked upon each other if there is no required intervening parameter. For instance, the –r switch reads a pcap file, in this example arp.pcap, and requires a file name to directly follow. The –n switch disables port and hostname resolution. This means that tcpdump uses port number instead of the equivalent service name (port 80 instead of HTTP) and does not perform DNS resolution on the IP address. Performing DNS resolution on a sizeable pcap is highly discouraged because this takes much more time due to the DNS lookups for IP addresses. The –e option displays the link layer, in this case, Ethernet. By default, tcpdump does not display the link layer. Finally, the –t switch suppresses the output of a timestamp associated with the capture time. We're not interested in the time and it creates some visual clutter if you don't care about it.

ARP is specified by RFC 826 in case you want to read more about it.

ARP is replaced by something known as Neighbor Discovery Protocol (NDP) in IPv6. NDP is a special ICMPv6 type and protocol. We'll cover this in more detail in the IPv6 module.



This is an ARP request as decoded by Wireshark. The Ethernet II and ARP protocols have been expanded to show you the fields within each. The Ethernet II header details that the sending MAC address is aa:00:04:00:0a:04 and the destination MAC address is the link broadcast of ff:ff:ff:ff:ff:ff. The "Type" field in the Ethernet link layer header indicates that the ARP protocol follows designated by the Ethernet type of 0x0806.

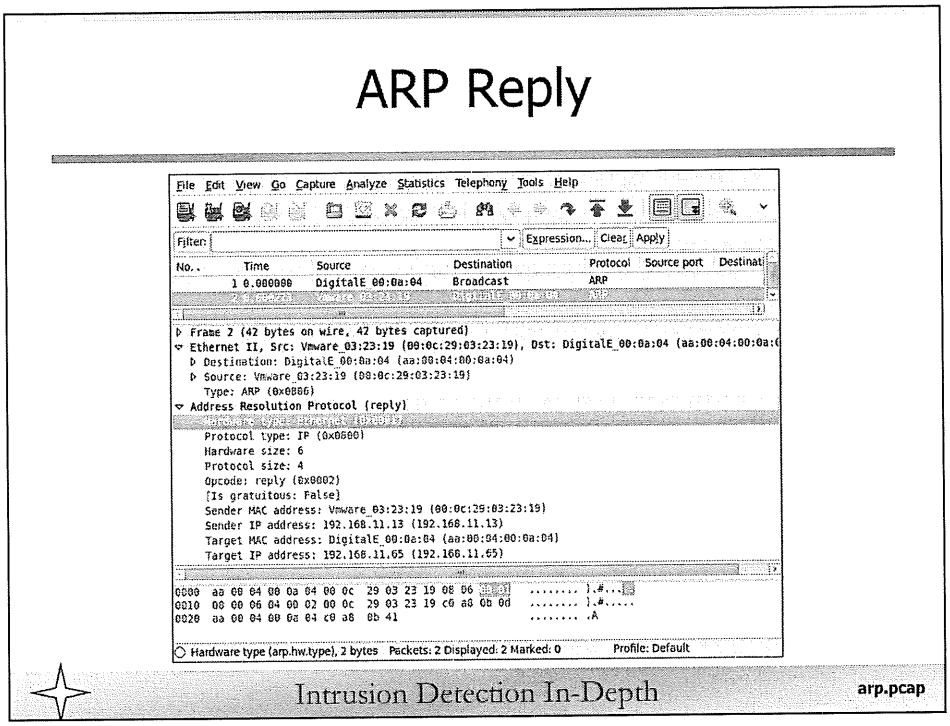
If we examine the ARP layer that follows, we note that this is a request – signified by the opcode of 0x0001; you see both the sender's MAC address and IP address of 192.168.11.65. The destination or target IP address is 192.168.11.13 and since the destination MAC address is unknown, a place holder value of 00:00:00:00:00:00 is supplied.

All the listening hosts will either update or place a new entry of the IP and MAC address pairings of the requester 192.168.11.65.

To examine the ARP request, enter the following on the command line:

wireshark arp.pcap

This is the first record in the pcap. Expand the Ethernet II and ARP protocols to see all fields.

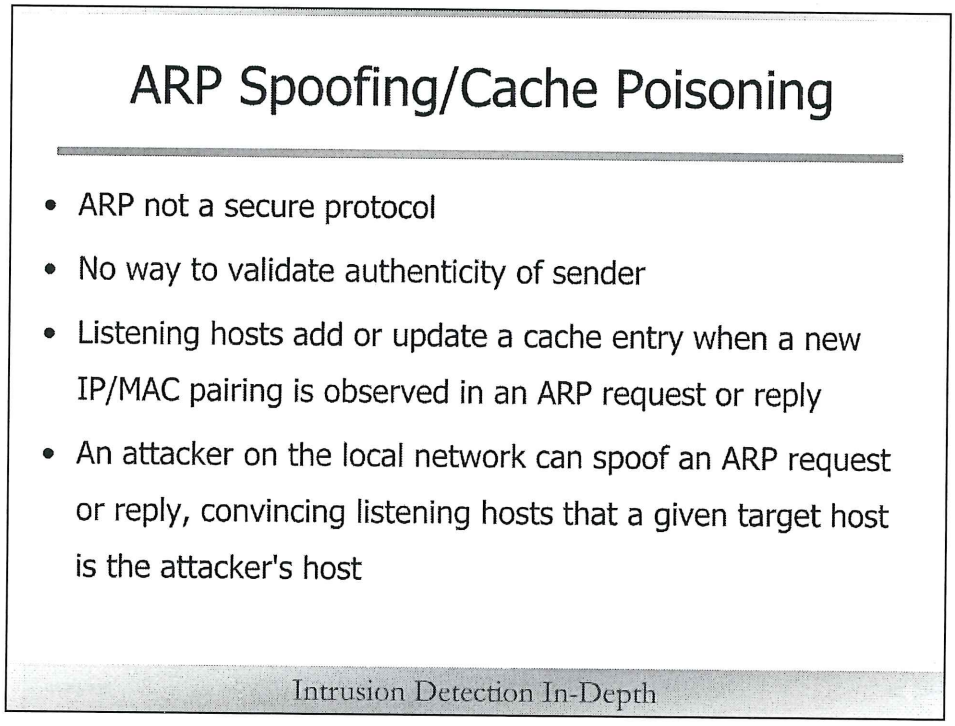


This is the ARP reply that ostensibly is from 192.168.11.13. It reports a MAC address of 00:0c:29:03:23:19. 192.168.11.65 and any other host that is able to see this traffic will cache the IP and MAC address pairings.

To examine the ARP reply, enter the following on the command line:

wireshark arp.pcap

This is the second record in the pcap. Expand the Ethernet II and ARP protocols to see all fields.

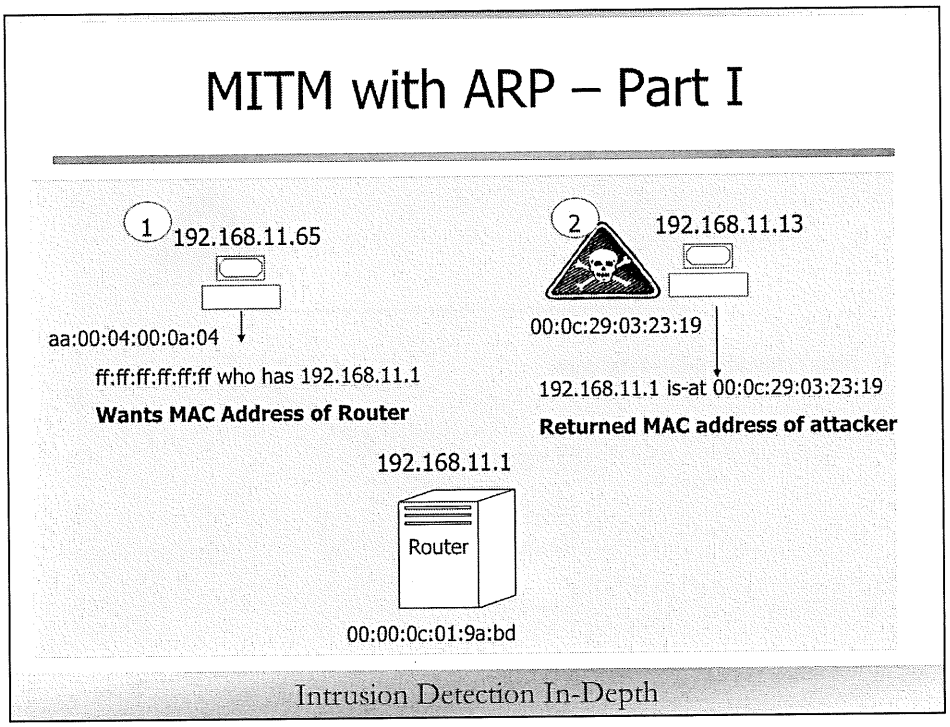


As you have most likely surmised by now, ARP is not a secure protocol. There is no way to validate that a host issuing an ARP request or reply is the host it alleges to be. In other words, if a malicious host on the network can convince other hosts that it is the owner of a given MAC address, the attacker's host can pose as another host or a local router.

ARP is stateless and naive. A host will cache a MAC address received in an ARP response even if it did not make the request. As if this is not bad enough there are some more weaknesses with the protocol that exacerbate the problem. Remember that ARP requests are made to the broadcast address making it a noisy and disruptive protocol. Eventually, ARP cache entries will expire, but until they do, unnecessary duplicate requests will not be issued. As well, ARP cache entries will be overwritten when a new ARP reply is seen.

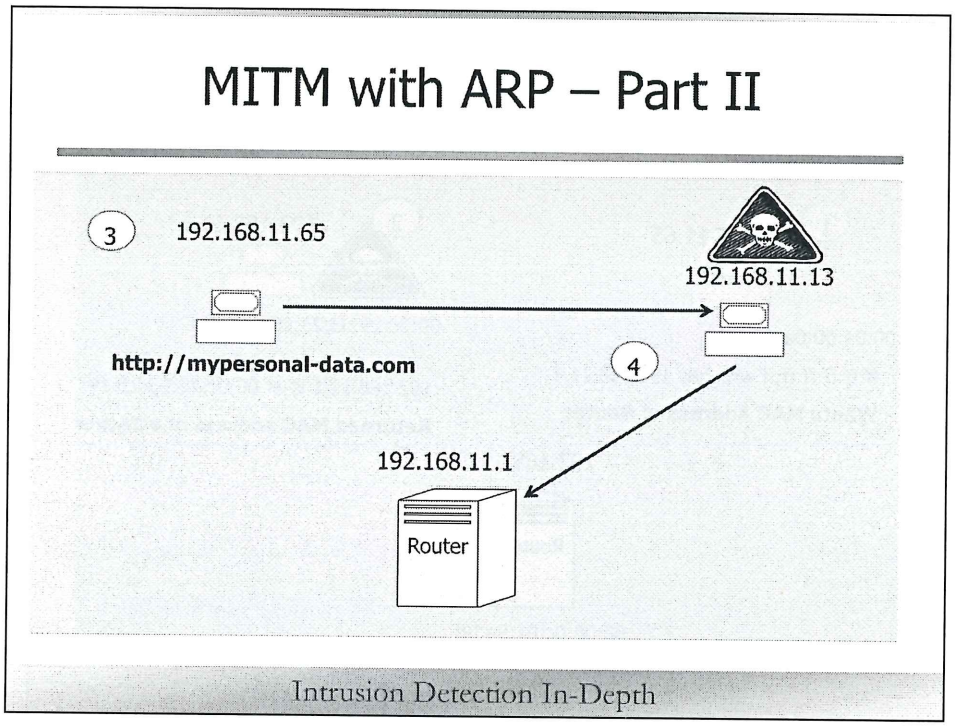
All of these conditions combine to make it very easy for a host to pose as another host or even a router. If this can be accomplished, a man-in-the-middle (MITM) attack may be possible, where a host poses as another node on the network, receives the traffic – examines it or alters it and forwards it to the eventual destination. The malicious host can process the traffic again before being returned to the original sender.

Most carrier class switches have some kind of ARP spoof protection mechanism by tracking MAC and IP address pairings and discarding invalid ones that do no match existing binding entries.



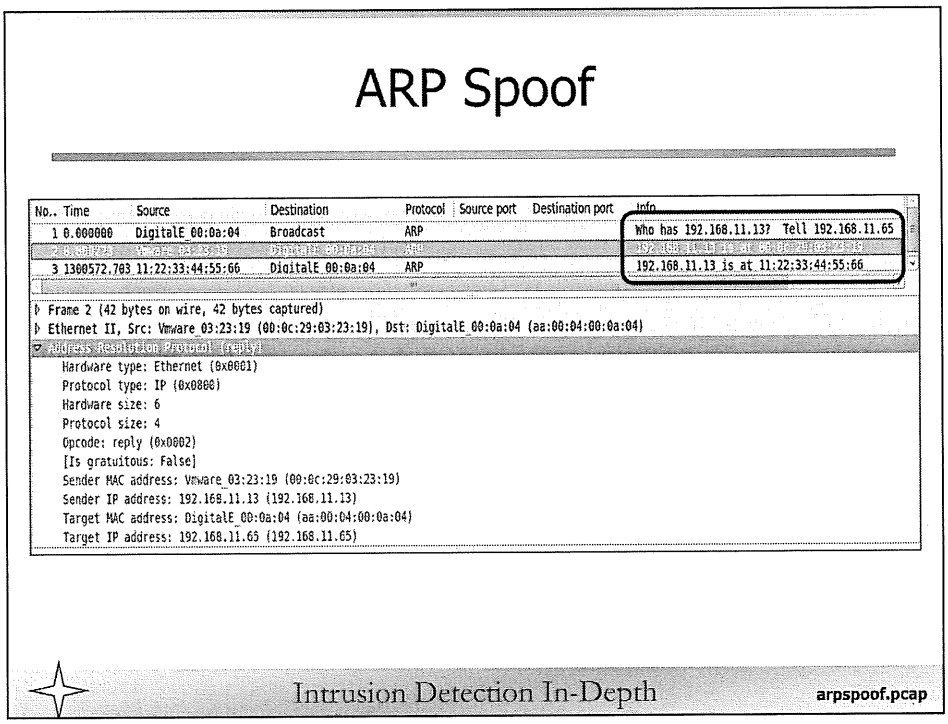
Suppose 192.168.11.65 uses router 192.168.11.1 for the next hop device for its traffic. Now, suppose that 192.168.11.65 does not have the MAC address of 192.168.11.1 in its cache. In step 1, it issues a link layer broadcast for the MAC address of the router. All hosts on the broadcast network can see this ARP request. Further suppose that there is a malicious actor on the local network at host 192.168.11.13. It sees the ARP request and in step 2 responds to 192.168.11.65 that the MAC address for the router is 00:0c:29:03:23:19. But, that is the MAC address for 192.168.11.13 – not the router.

Oh, no! 192.168.11.13 has performed an ARP spoof and managed to poison the ARP cache of 192.168.11.65 and any other host that caches this bogus pairing. Hosts may update their cache before an entry expires. If this is true about the target host, the attacker must issue the bogus ARP reply after the router's reply.



Now, in step 3 192.168.11.65 wants to go to a site that stores some personal data on a web server. The traffic is not encrypted so it can be seen in the clear. In preparation for traffic from 192.168.11.65 the attacker on 192.168.11.13 has put her/his host in IP forwarding mode. This reads packets from the network and forwards them as in step 4, in this case, to the router 192.168.11.1 that 192.168.11.65 wanted to find.

Now, the attacker can read all the traffic from 192.168.11.65. As well, all traffic that is returned via 192.168.11.13 if the attacker manages to convince the router that the attacking host has the MAC address of 192.168.11.65. An alternative is to set up some kind of proxy on 192.168.11.13 so that the outbound traffic carries the IP address of 192.168.11.13 and the return traffic is automatically sent to it without having to poison the router's ARP cache. The traffic is then returned to 192.168.11.65 – perhaps intact, or perhaps altered either outbound or inbound.

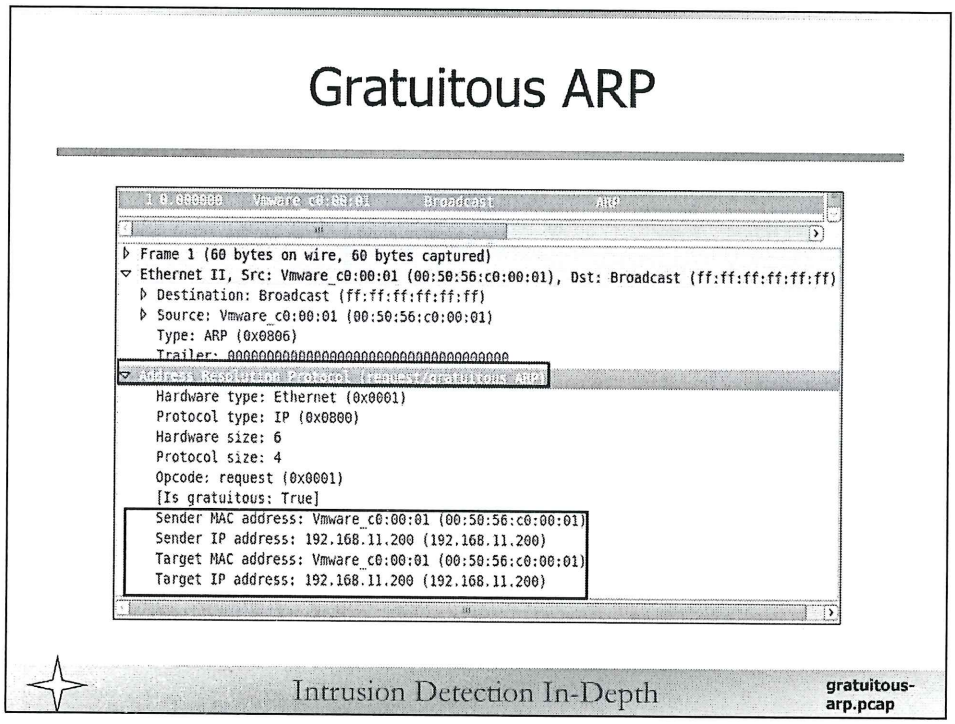


Let's look at an example of a spoofed ARP reply. The first record reflects an ARP request from 192.168.11.65 for the MAC address of 192.168.11.13. The legitimate reply follows in record 2. The host 192.168.11.13 replies that its MAC address is 00:0c:29:03:23:19. In record 3, a spoofed reply comes from a host alleging to be 192.168.11.13, yet having an associated MAC address of 11:22:33:44:55:66. This particular bogus MAC address is used to make the ARP spoof/poisoning more obvious.

If host 192.168.11.65 caches the MAC address on the spoofed ARP reply, its cache is poisoned with this bad pairing. Even though the legitimate ARP data has not yet expired, the cache is updated with the poisoned paring. Now, when 192.168.11.65 wishes to send traffic to 192.168.11.13, it will really be sending it to the host with the MAC address of 11:22:33:44:55:66.

To see the above output:

You should be in Wireshark already. Just open a new file by navigating to the File→ Open menu and selecting file **arpspoof.pcap** from the appropriate directory.



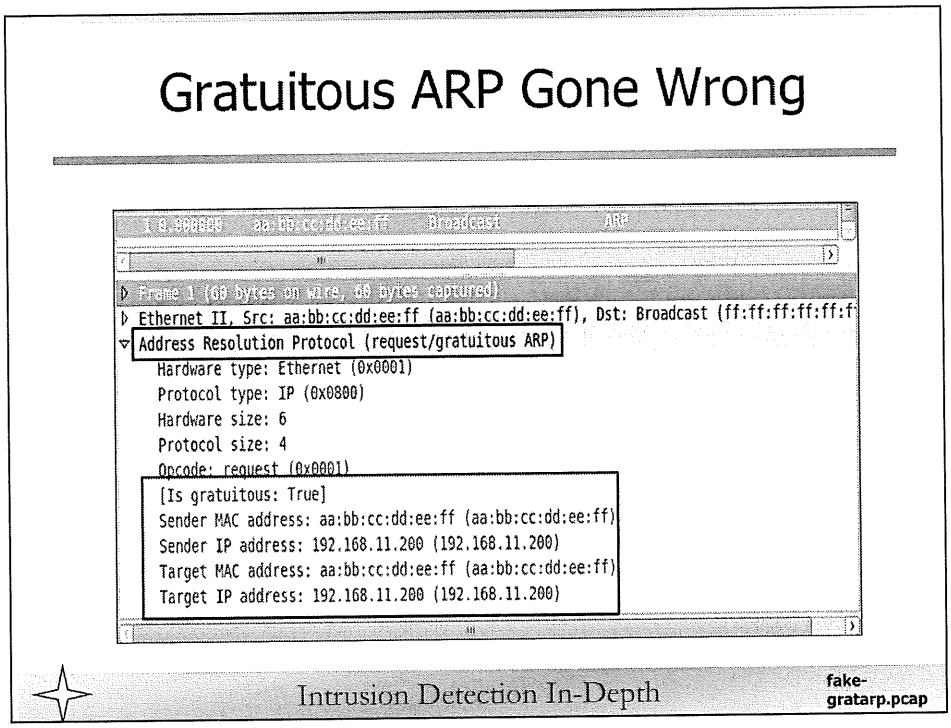
There is a particular ARP known as a gratuitous ARP. This is when a host sends an ARP request to the broadcast address, like a normal ARP request, yet the sender and target addresses are the same. A gratuitous ARP asks for the MAC address for its own IP address. This accomplishes broadcasting it's IP and MAC pairing to all the hosts on the network. Sound strange?

There are a couple of situations when a legitimate gratuitous ARP is used. The first is to determine if there are duplicate IP addresses. If a legitimate response is received, there is a duplicate IP address. This should be resolved by the system administrator. The second situation is where a host receives a new network interface card with a new MAC address. The gratuitous ARP serves to notify all hosts to cache the new IP address and MAC address pairing, possibly overwriting the old one, if cached.

How can you tell a gratuitous ARP from a regular one? Well, Wireshark designates it as gratuitous, but there are some unique characteristics that define it as such. Both the Sender and Target MAC address fields in the ARP request portion of the packet are the same as are the Sender and Target IP address. A regular ARP request will have a Target MAC address of 0.0.0.0 and a Target IP address that is different from the Sender IP address.

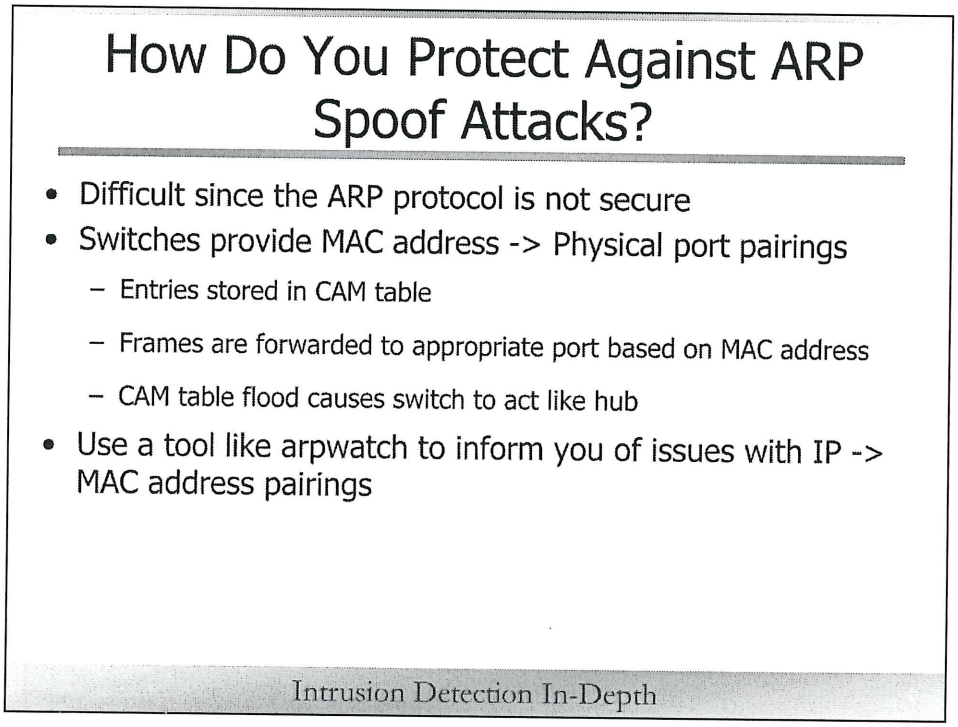
To see the above output:

You should be in Wireshark already. Just open a new file by navigating to the File→ Open menu and selecting file gratuitous-arp.pcap from the appropriate directory.



The Wireshark output in this slide shows a gratuitous ARP request. This can be a real gratuitous ARP or it can be an ARP poisoning attack causing listening hosts to cache false MAC and IP address pairings. The ARP reply above professes to be from host 192.168.11.200 and to have a MAC address of aa:bb:cc:dd:ee:ff. This particular bogus MAC address is used to make the ARP spoof/poisoning more obvious.

You should be in Wireshark already. Just open a new file by navigating to the File→ Open menu and selecting file **fake-gratarp.pcap** from the appropriate directory.

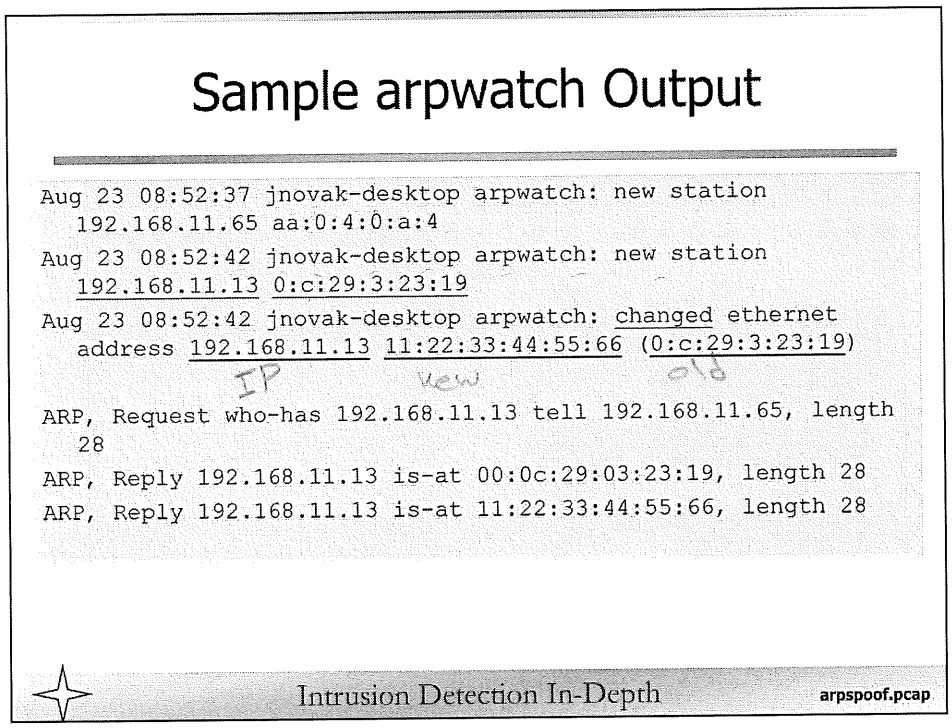


ARP cache poisoning is a simple nasty, yet very effective attack. As we've seen ARP is not secure; there is no validation of the authenticity of the sender. In addition, those hosts not directly involved in a given ARP request/response exchange, naively add or update cache with observed pairings in ARP packets. It's almost as if the ARP specifications encourage bad behavior.

A hardware switch may offer some protection. Switches store MAC address to physical switch port pairings. This means that when a frame is received, it is forwarded to the physical switch port that has that destination MAC address associated with it. This is what differentiates a switch from an unintelligent hub that forwards all packets out all ports. The switch stores the MAC address and port pairings in something known as a Content Addressable Memory (CAM) table stored in memory. However, there is an attack where a switch receives a flood of MAC addresses, filling up the CAM table. After this occurs, an attacker can circumvent the built-in protection of port to MAC address pairing by sending a spoofed ARP packet. Also, a successful attack on the CAM table causes the switch to act like a hub by sending all frames to all switch ports instead of exclusively to the actual port associated with the destination MAC address. This can permit the attacker to see, sniff, and possibly alter traffic that she otherwise could not.

The point is that it is possible for the pairing of switch ports to observed MAC addresses protection mechanism to be overridden. However, there are countermeasures such as port security that restricts the association of a port with a single source MAC address. Additionally, some switches can be configured to disable sending frames to all switch ports. Most enterprise switches have protection against ARP spoofing.

The good news is that the attacker must already be in the network to perform any of these ARP attacks. Yet, the bad news is that it is relatively easy to poison a host's ARP cache if security mechanisms are not present. There are tools like the venerable arpwatch that sit on the local network and track the MAC → IP pairings. If anything seems amiss, arpwatch will log a message for further processing to alert the analyst.

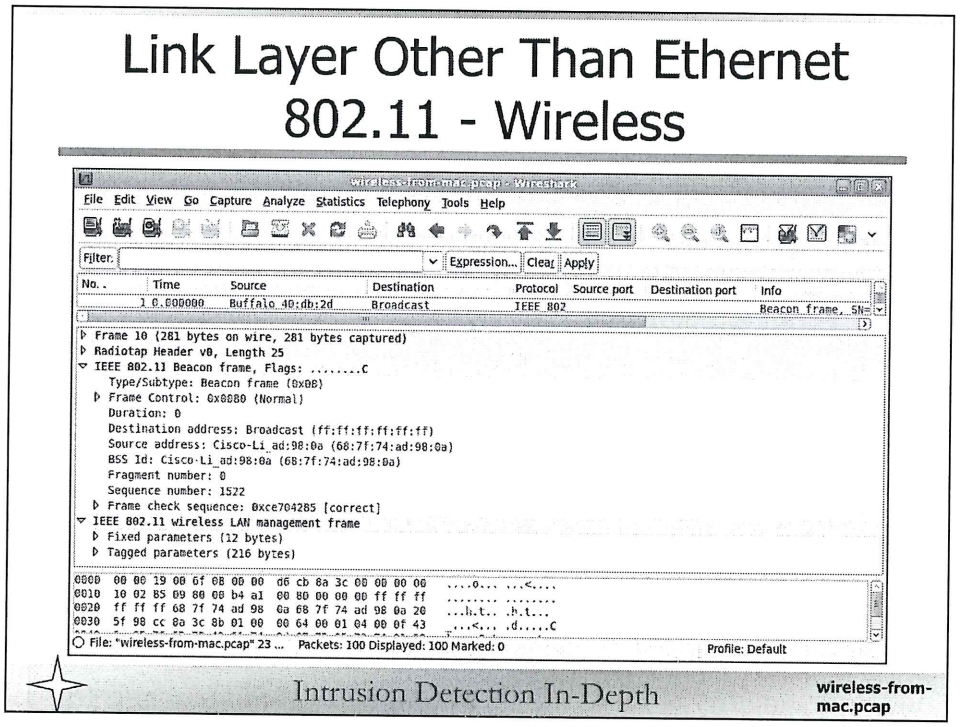


This is a sample of the type of output you see from arpwatch as it observes the traffic shown as tcpdump records on the bottom of the slide. The first entry notes the presence of a new host on the watched network. The new host is 192.168.11.65 with a MAC address of aa:00:04:00:0a:04. A second host appears on the network with an IP address of 192.168.11.13 with a corresponding MAC address of 00:0c:29:03:23:19. Everything is fine so far.

Now an ARP reply appears on the network where IP address 192.168.11.13 professes to have a MAC address of 11:22:33:44:55:66 . This appears to be a spoofed MAC address with an attempt to poison the cache of 192.168.11.65 or any other host that sees the ARP reply. The MAC address 11:22:33:44:55:66 is ostensibly the MAC address of the attacker.

The tcpdump output that caused the generation of the arpwatch messages is shown. First 192.168.11.65 requests the MAC address of 192.168.11.13. The first reply, we assume, is from the real host, although we cannot tell just by looking at the output that this is the real host. We'd have to manually investigate the network hosts to determine whether the first or second ARP reply is the real one. Now, a second ARP reply is sent indicating that host 192.168.11.13 is at MAC address 11:22:33:44:55:66. When 192.168.11.65 sends traffic to what it believes is 192.168.11.13, the traffic is actually going to the host associated with MAC address 11:22:33:44:55:66 – we suspect the attacker's host. The unrealistic MAC address of 11:22:33:44:55:66 is used in this demonstration to make it more obvious which is the spoofed ARP reply.

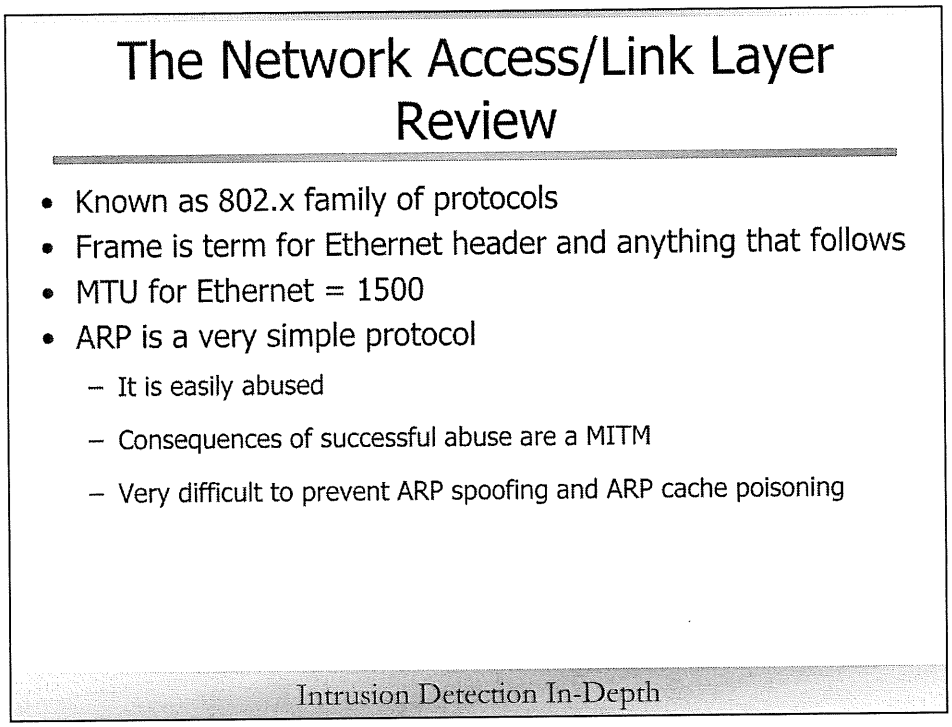
- The tcpdump output can be generated with the command:
  tcpdump –r arpspoof.pcap -nte.



Remember that there are other 802.x link layer protocols such as 802.11 wireless. Capturing 802.11 packets for data is typically not done since the traffic should be encrypted. However, there are still clear text management frames in 802.11 traffic that can provide some insight about the packet. Beacon frames show the advertisement by a wireless access point of its Service Set Identifier (SSID), probe requests can indicate an attempt to scan the wireless traffic, authentication frames can be used to indicate brute-force attacks when found in abundance, deauthentication frames, association and dissociation essentially register and unregister with the wireless access point, recording each mobile device for proper frame delivery.

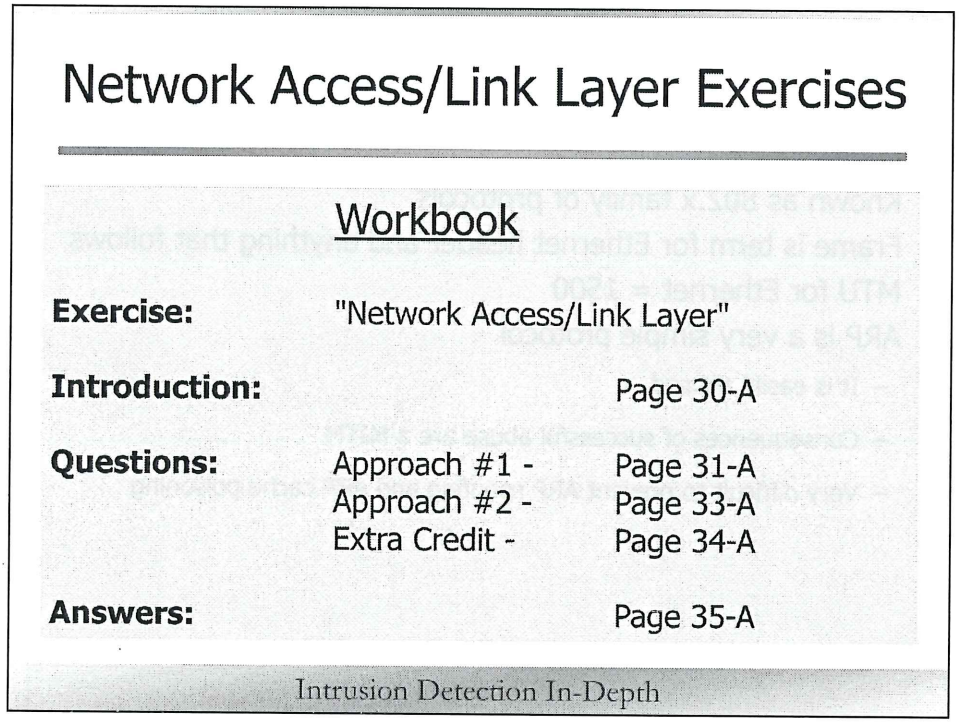
The Wireshark display shows an 802.11 beacon frame, consisting of an IEEE 802.11 header followed by the management beacon data.

You should be in Wireshark already. Just open a new file by navigating to the File→ Open menu and selecting file **wireless-from-mac.pcap** from the appropriate directory.



The link layer uses the 802.x family of protocols overseen by IEEE. In this course, we examine Ethernet 802.3 link layer, however there are other 802.x link protocols such as wireless and Bluetooth. An entity that begins with an Ethernet header is known as a frame. The MTU for Ethernet is 1500 bytes – meaning that is the maximum size the link layer can handle of data following the Ethernet header. The Ethernet header is 14 bytes, but not included in the maximum 1500-byte frame size. The smallest frame size is 64 bytes, including the CRC. When the length of the Ethernet header and subsequent data such as the IP header and its payload is less than 60 bytes, it must be zero-padded to be 60 bytes.

We learned that ARP is the means by which the IP layer discovers the link layer address for a given IP address. This association allows the two layers to communicate. However, ARP is pretty simple and easily spoofed, resulting in ARP cache poisoning. This allows the attacker to create a MITM going through the attacker's computer, permitting the traffic to be viewed or perhaps altered. It is almost impossible to prevent ARP spoofing, however there are tools like arpwatch that can assist you in discovering attempts of ARP cache poisoning.



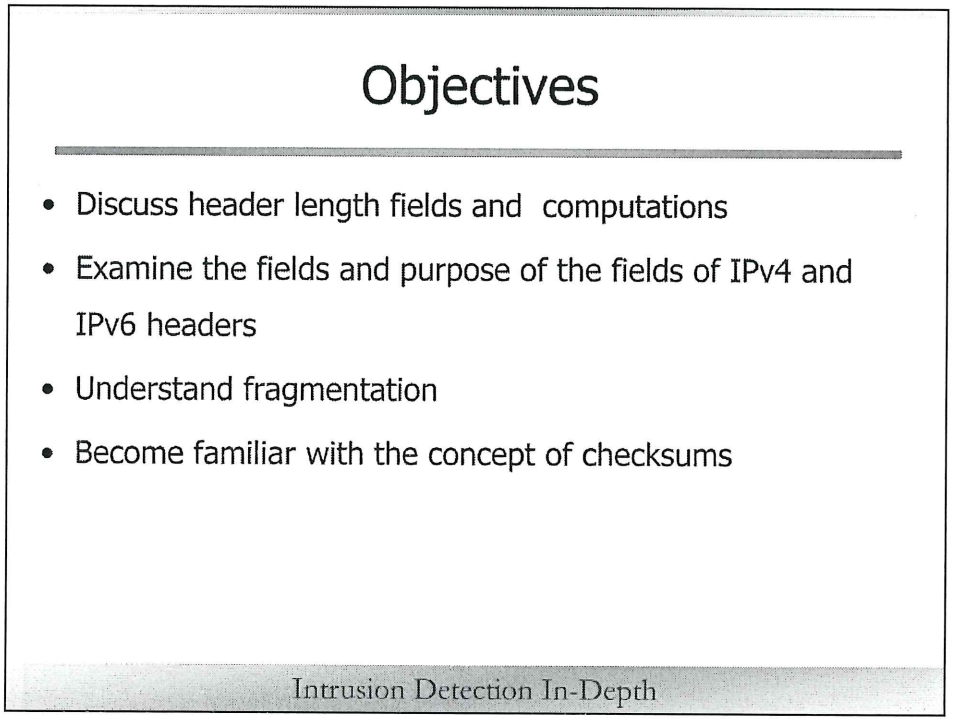
This page intentionally left blank.

The IP Layer

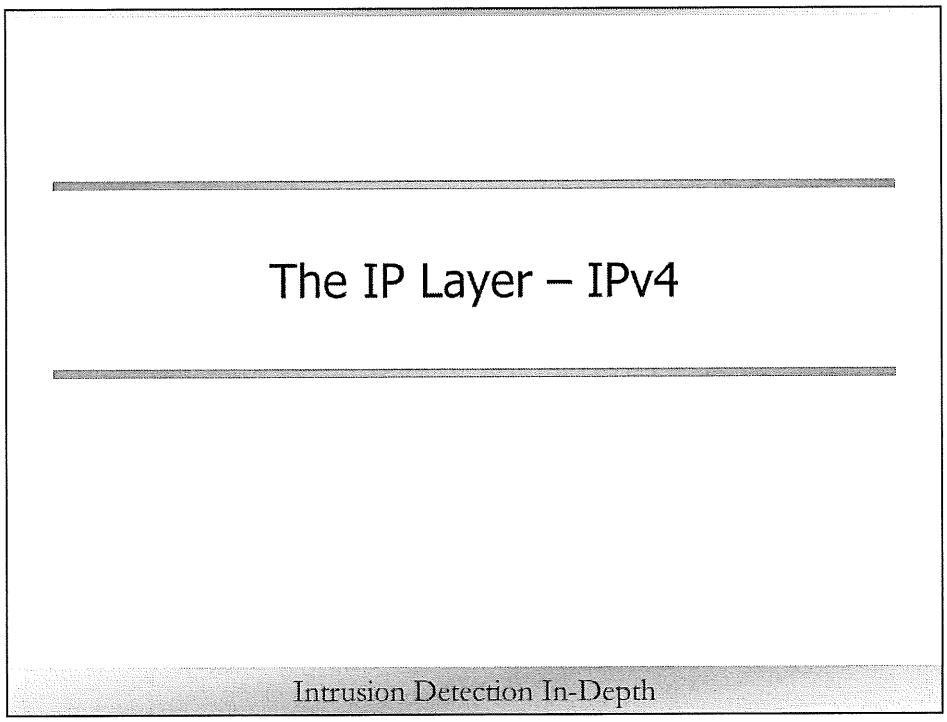
- Concepts of TCP/IP
- Introduction to Wireshark
- The Network Access/Link Layer
- **The IP Layer**
  - IPv4
  - IPv6

Intrusion Detection In-Depth

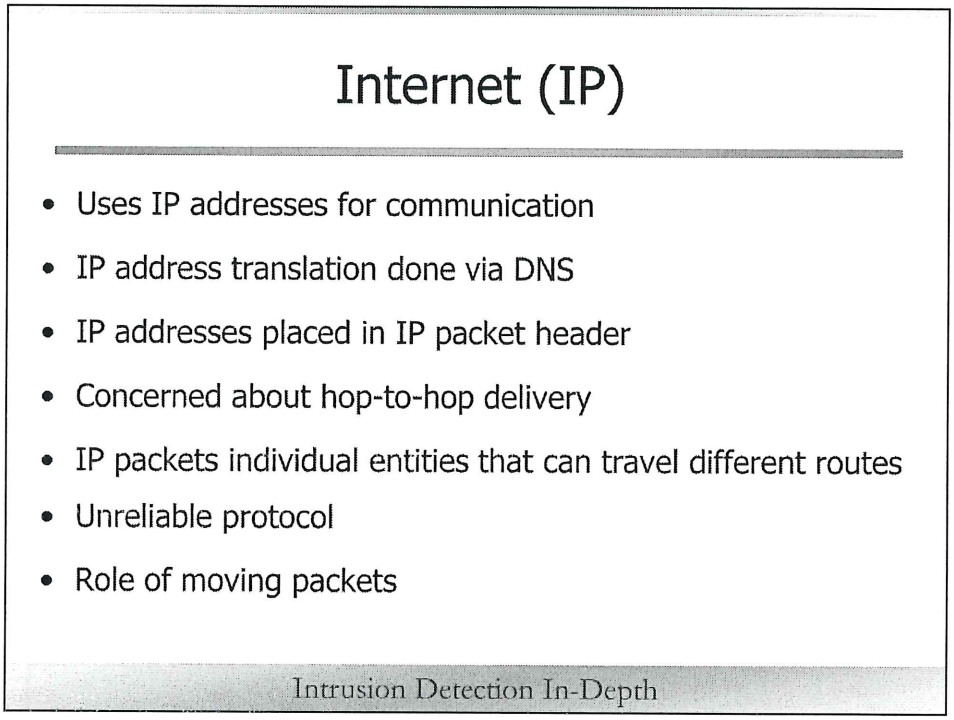
This page intentionally left blank.



The IP layer is responsible for getting packets from hop-to-hop. We'll examine the IPv4 and IPv6 header values and formats for how this is accomplished. Fragmentation is a result of splitting an oversized IP packet into one or more packets. We'll see how this is performed and the packet formats that result. Another concept that is important to know is checksums. The checksum is a computation done on a given header and perhaps data to detect any corruption that may have occurred in transit.



This page intentionally left blank.

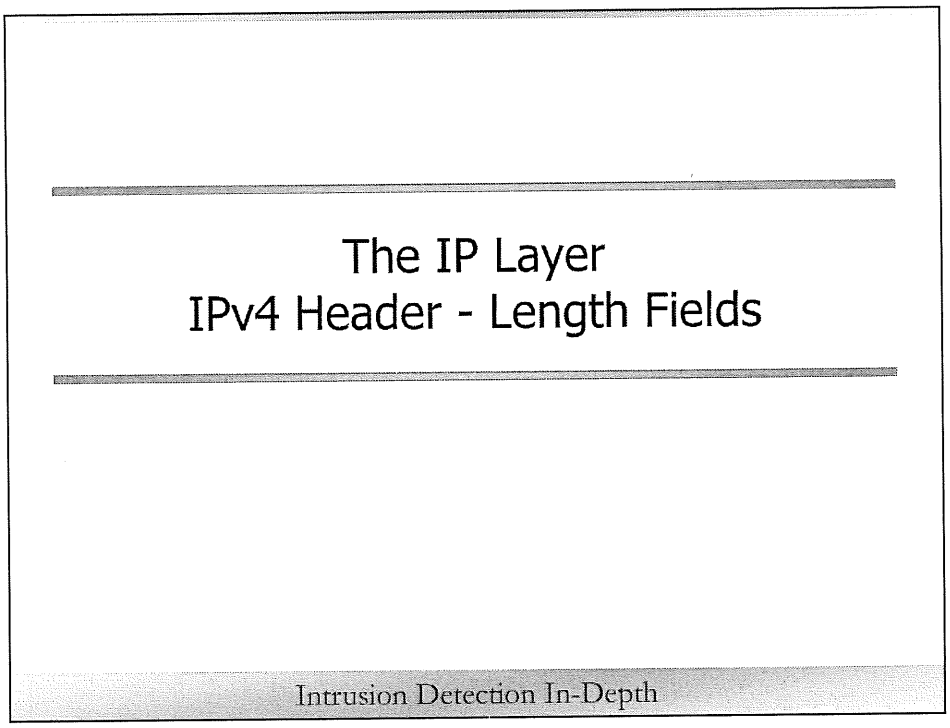


The Internet layer uses source and destination IP addresses to communicate between hosts. As users, we tend to refer to hosts by their hostname rather than their IP addresses. Usually, we can remember hostnames more easily than IP addresses and we rely on the Domain Name Server (DNS) to make the translation between hostname and IP address and vice versa. We'll explore DNS very thoroughly later in the course.

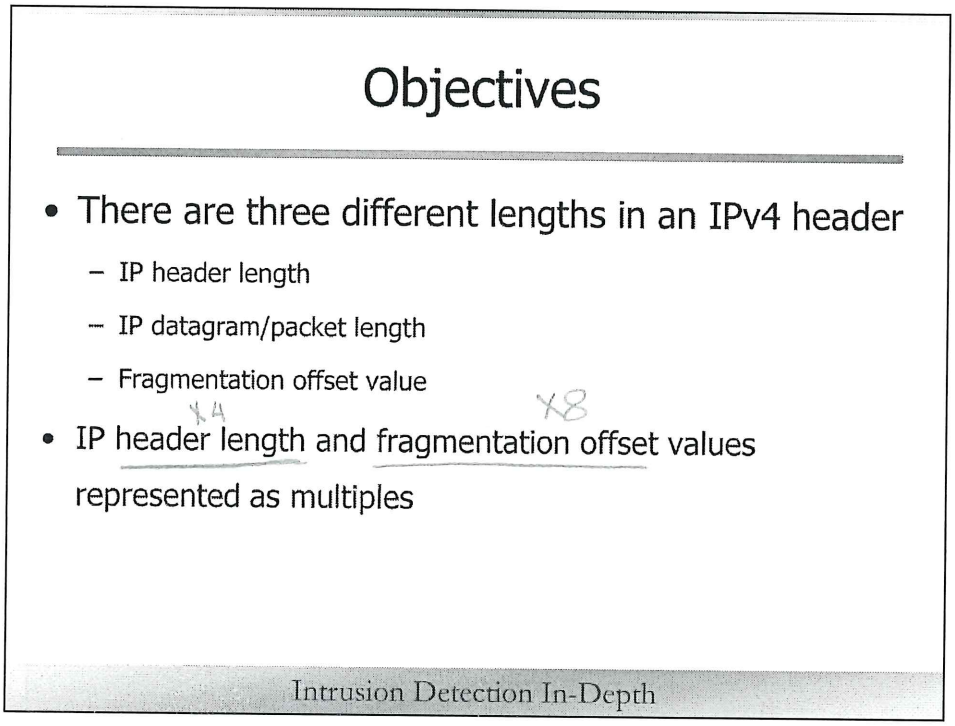
The IP addresses are stored in the IP header of the IP packet. IP is concerned about transmitting a packet from hop-to-hop. These IP packets are individual entities that are directed to the destination host as they move. It is even possible for different IP packets that share the same source origin and sent to the same destination IP's to travel different routes. There may be a change in routing based on the dynamics of the Internet. It is possible that a router has gone down or that different routes become optimal ones. The real role of the Internet layer is to move packets.

It is important to understand that IP is not a reliable protocol. It makes no guarantees about delivery of the packet. The packet can get lost, or expired, or dropped and IP will not know and will not care. That sounds kind of heartless, but that is not the function of IP. Reliability is the responsibility of the transport protocol or the responsibility of an application that will notice the packet loss and try to rectify the problem.

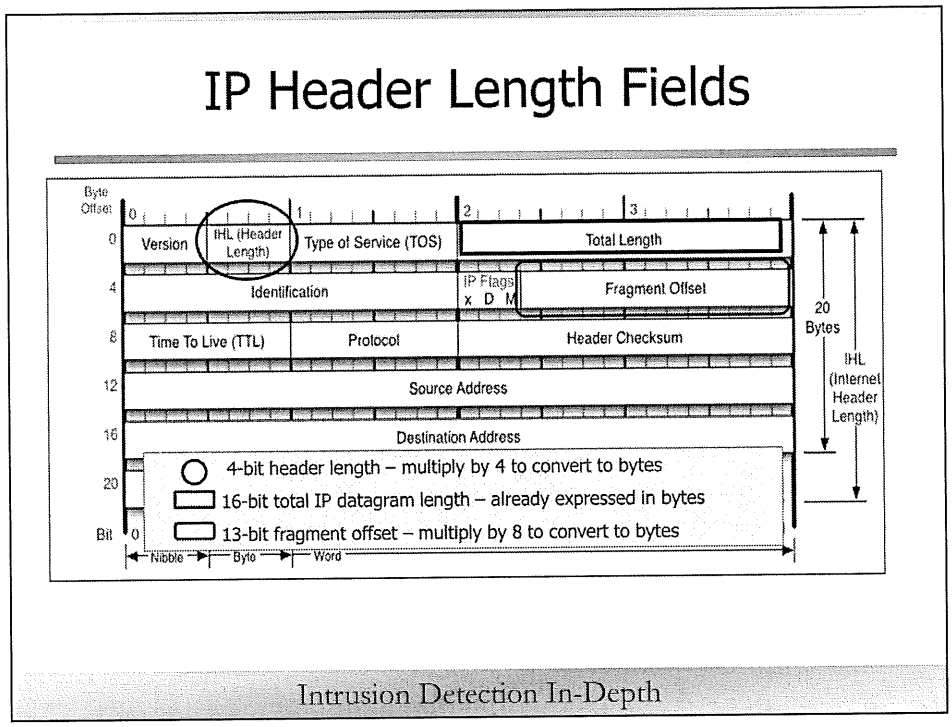
RFC 791 discusses IP in much more detail.



This page intentionally left blank.

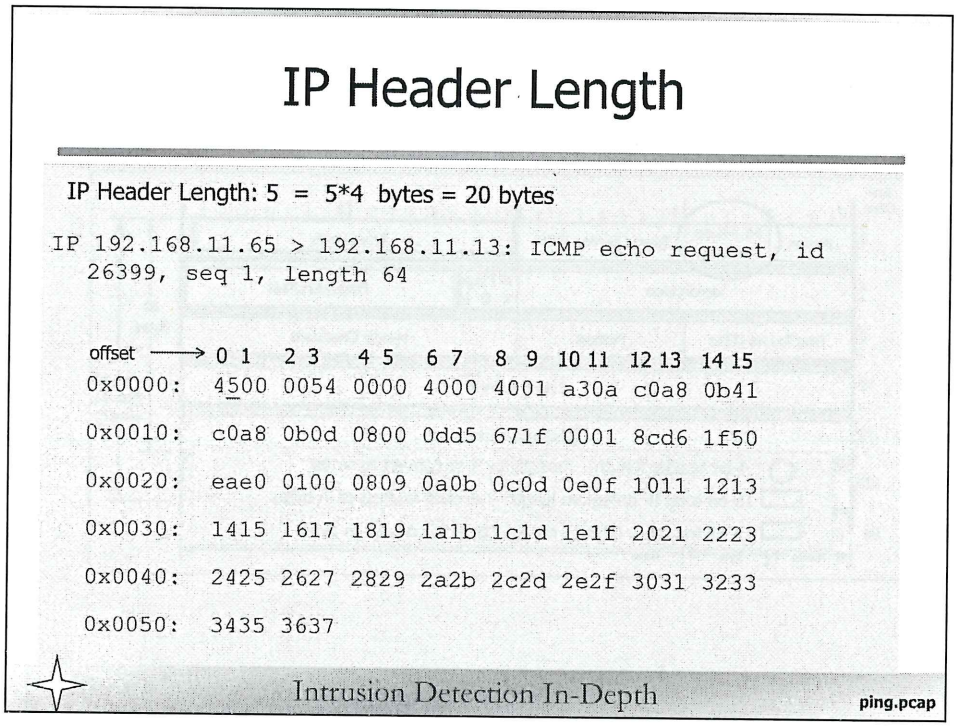


Remember early in the first section of today that we discussed how de-encapsulation required most protocols to have length values to know where a given header/data stopped and another started. The IP header has three different length fields required for determining this. The IP header length and the IP datagram length are used for all traffic, whereas the fragmentation offset only for fragments. The IP datagram length is straightforward and expressed as a normal value. The IP header length and fragmentation offset values require some explanation as the values found in the IP header must be multiplied by a fixed value to determine the actual value.



The IP header has to indicate lengths of different aspects of IP. Some of these length fields are straightforward where the value in the field is the number of bytes of length. However, there are others that are represented as a multiple of a certain value. If you look at the IP header above, you'll see that there are three different fields containing lengths. None of these fields except the 16-bit IP datagram total length is the actual byte length of the field. Let's examine these fields in more detail.

Many of these length fields are used to break the packet down. When you get a hex dump of a packet that you'd like to analyze without a tool, it is pure nonsense unless you can figure out where headers stop and embedded protocols or payload begin. For instance, you always have to be able to figure out where the IP header stops and the embedded protocol begins after it since the IP header can be variable in length if there are IP options. All of this is necessary to interpret the actual meaning of the packet.



The first of the length fields is the IP header length that includes the IP header only. Most IP headers have a standard length of 20 bytes, however it is possible to have a packet with options such as source routing that require more space for the IP header. This means that one of the first steps in de-encapsulation of a datagram is to calculate the IP header length.

Remember that when you are dealing with hex bytes that counting always begins with zero. We often need to express a field by its offset from the beginning of the data under inspection.

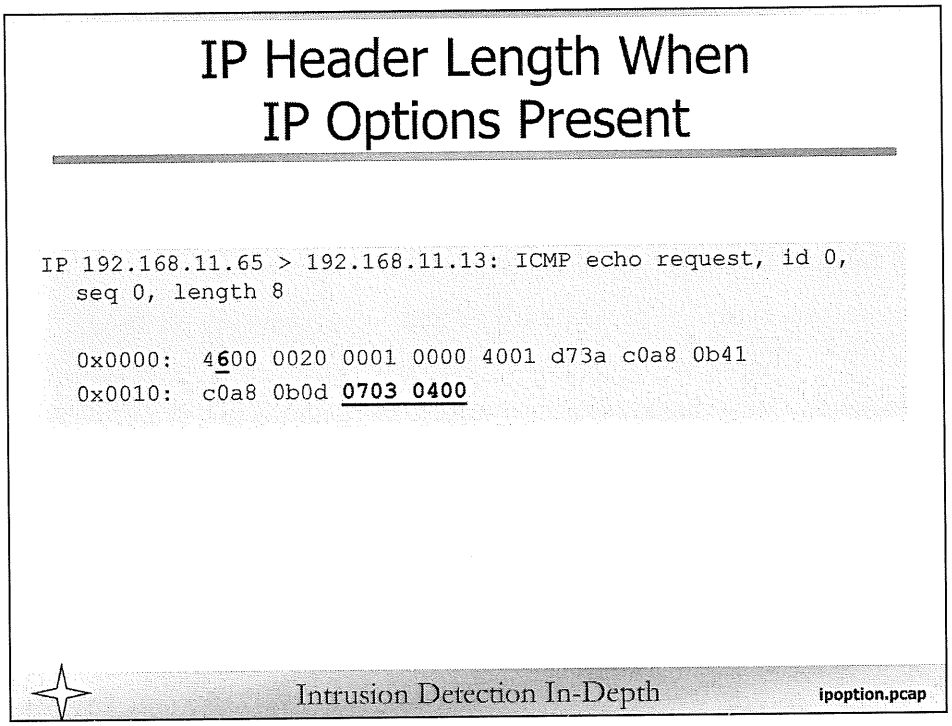
The IP header length is found in the low-order nibble (4 bits) of the zero byte offset into the IP header. In the above slide, we see that the IP header length is 5. This is not 5 bytes as one might assume. This is actually 5 32-bit/4-byte units. Years ago, this entity was called a "word" and was associated with the processor design of the time. What this means for you is that you have to use a multiplication factor of 4 to figure the actual number of bytes. In this case, we see that we have 20 bytes. Since this field is 4 bits long, the greatest value that can be found in it is a binary 1111 or a hexadecimal 0xf which is a decimal 15. This means the longest IP header can be 60 bytes (15*4).

You may be wondering why you have to go through this conversion – why didn't they just make the field long enough to express in bytes? That would require 2 additional bits ($2^6 = 64$) to represent the maximum of 60 bytes. This would require every IP datagram to be 2 additional bits longer – increasing the amount of data in the IP header. While an additional 2 bits are practically inconsequential today, this was not the case when TCP/IP was conceived years ago when there were much slower wire speeds and architectures.

To view this output, enter from the command line:

tcpdump –r ping.pcap –ntx –c1

The –c1 option tells tcpdump to show a single record – count 1.



Now, let's look at a datagram that has IP options. If there are IP options, the header length will be greater than the standard 20 bytes. Examining the header length field, we see that it has a hex value of 0x6. Multiplying that by 4, we know that the IP header length is 24 bytes. If you count the underlined bytes representing the IP header, you'll see that indeed it is 24 bytes.

Look at the type of IP option that is in this datagram found in the 20th byte offset. This is convention to place the first IP option in the 20th byte offset of the IP header. Each of the various IP options, such as loose and strict source routing, has a different one byte representation .

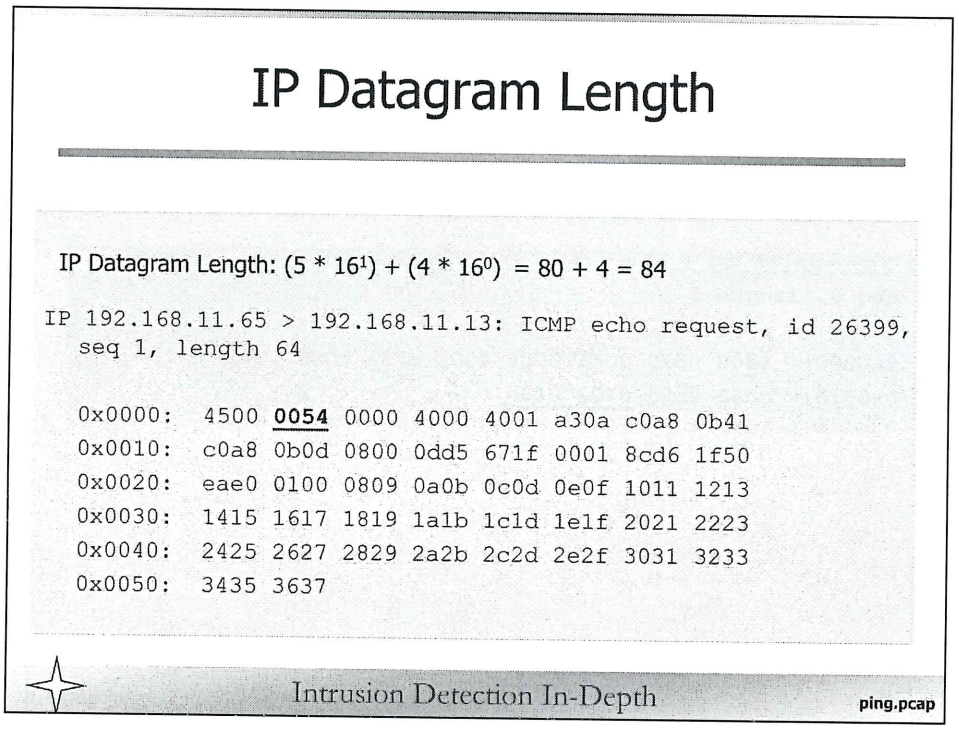
See http://www.iana.org/assignments/ip-parameters to see the possible different IP options.

In the above output, there is a hex value of 0x07 in the 20th byte offset, indicating an option known as record route. This attempts to collect IP addresses for all routers through which the datagram travels. Each route IP takes up 4 bytes. The record route option itself has 4 bytes in the IP header as overhead. This means if the maximum IP header is 60 bytes and we must have 20 bytes for the standard header, we only have 40 bytes left for recording IP addresses. This allows 9 timestamps to be collected which may not be enough to record all router IP's through which the datagram travels. This is basically what a traceroute finds, except as we'll learn tracertoute uses a transport layer protocol to determine the routers. Some sites block traffic that attempts to assess the network so it is possible that this type of option may not make it to its destination.

The hex dump shows an ICMP echo request with a header length of 24 bytes. The standard header ends at offset 19 and the 4 bytes of remaining IP options directly follow. All of the IP header bytes are underlined. The ICMP message header and data follow.

To view the output, enter in the command line:
tcpdump –r ipoption.pcap –ntx

Where space permits, there may be multiple IP options following the IP header. Each is distinguished by an IP option type, a length value if the IP option is greater than one byte, and the accompanying value. The length field is used to find the end of the current IP option and the beginning of the next, if one follows.



The IP datagram length is straightforward. You simply move to the 2nd and 3rd bytes offset of the datagram and convert from hex to decimal to see how long the expected length will be. The IP datagram length is the length of the entire packet including the IP header, the transport header, and any data.

The IP datagram length can be used to derive the length of the transport protocol data where the transport protocol header does not maintain that value. For instance, the UDP transport layer header has a length of the number of bytes in the UDP header and data that follows. However, neither TCP nor ICMP has such a value. As you may recall from a previous discussion the number of payload bytes for TCP or ICMP is computed as follows:

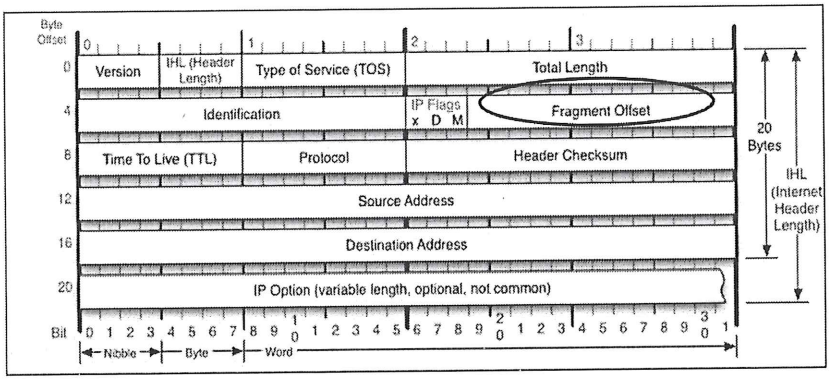
IP datagram total length – IP header length – protocol header length = data bytes

To view this output, enter from the command line:

tcpdump –r ping.pcap –ntx –c1

The –c1 option tells tcpdump to show a single record – count 1.

Fragmentation Offset Field

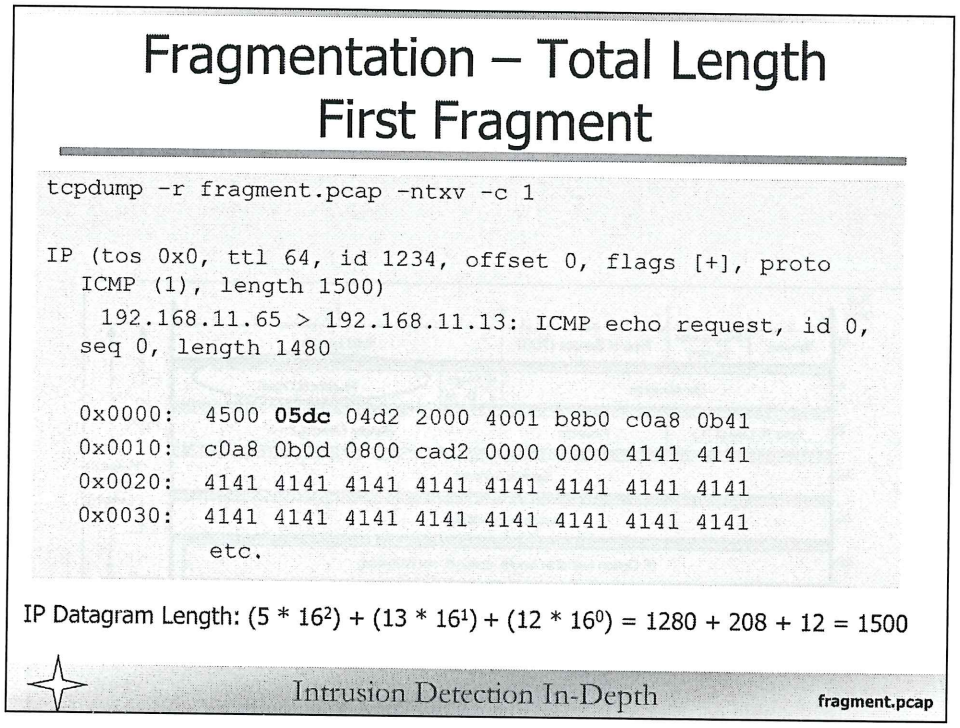


Intrusion Detection In-Depth

Looking at the above slide, the fragmentation offset field is found in the 13 low-order bits of the 6th and 7th bytes of the IP header. We'll cover the theory of fragmentation in detail in upcoming slides.

What you need to know right now is that fragmentation causes a single too-large packet to be divided into multiple packets. There is no guarantee that these packets will arrive at their destination in the same order in which they were sent. Nor is there a guarantee that all will arrive. Therefore, it is imperative that there is some kind of scheme to denote chronology – or where this fragment falls among all the others. This is specified by something known as a fragment offset found in every fragment of the entire fragment train.

This is why we use fragmentation offset



The tcpdump output shows the first fragment in a series of related fragments of a long ICMP echo request. The tcpdump options of –ntxv produced this output and serve to suppress name resolution and timestamp display, show the output in hexadecimal, and the –v option designates verbose in order to see more details in the first ASCII line about the fragment.

When a datagram becomes fragmented, the total length will change for each fragmented datagram. For all fragments but the last, this should be the MTU size of the link that caused the fragmentation – this case Ethernet with a size of 1500 bytes. Examining the tcpdump output for the fragment, we see that the fragment has a length of 1500 bytes according to the value in the 2nd byte offset from the IP header (IP packet length) and this is the first fragment of the fragment train since it has a zero offset.

If you look at the IP datagram length field, you see that we have a hex value of 0x05dc which computes to 1500 decimal. So, it appears that this packet was originally larger than 1500 bytes and needed to be placed on an Ethernet network. The 1500 represents the 1480 bytes of embedded fragment data plus the 20 byte IP header.

To view this output, enter at the command line:

tcpdump –r fragment.pcap –ntxv –c 1

This will show the output above as well as that seen on the next slide.

Fragmentation – Total Length Last Fragment

```
(tcpdump output from reading fragment.pcap continued)

IP (tos 0x0, ttl 64, id 1234, offset 1480, flags [none],
   proto ICMP (1), length 48)
    192.168.11.65 > 192.168.11.13: icmp

   0x0000:  4500 0030 04d2 00b9 4001 dda3 c0a8 0b41
   0x0010:  c0a8 0b0d 4141 4141 4141 4141 4141 4141
   0x0020:  4141 4141 4141 4141 4141 4141 4141 4141
```

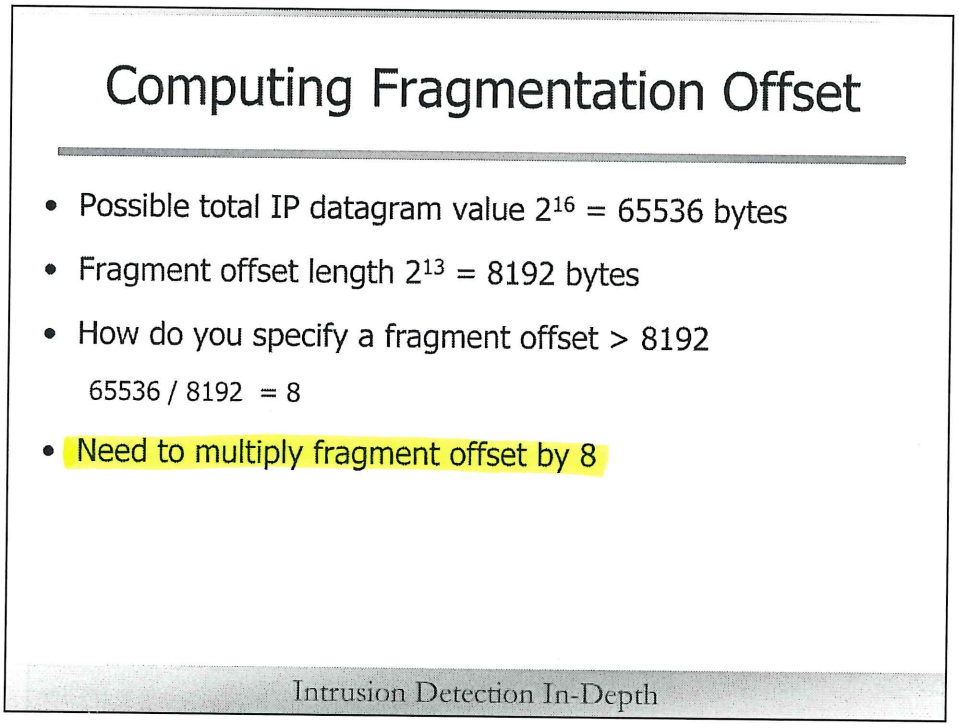
IP Datagram Length: (3* 16^1) = 48

This is the 2nd and last fragment associated with the fragment viewed in the previous slide. You can see that the total length of 0x30 is 48 bytes. This is 20 bytes of IP header followed by 28 bytes of data. The last fragment of a fragment train probably will not have the maximum 1480 bytes of data since it is whatever data is leftover from the previous fragment.

To view this output, enter at the command line:

tcpdump –r fragment.pcap –ntxv

This will show the output above as well as that seen on the previous slide.

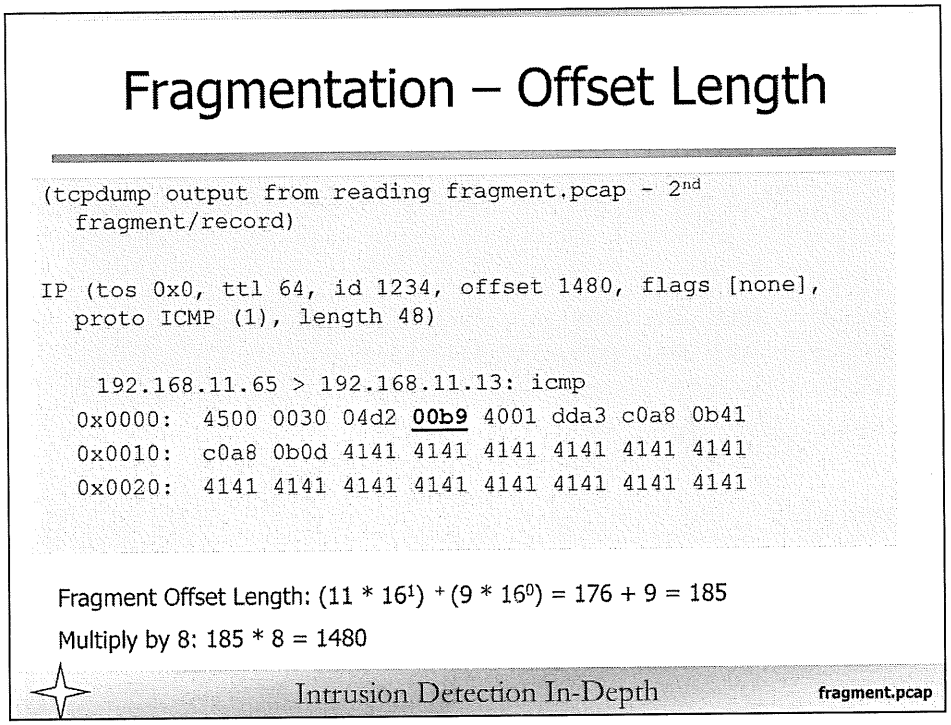


Theoretically, it is possible to have a datagram that is 65535 bytes long since the datagram length field is 16 bits. Given this, it is also theoretically possible that a fragment offset can be very close to this 65535 limit. But, the fragment offset field is only 13 bits with a possible maximum value of 8192 bytes. Therefore, some multiplication factor must be applied to the offset value in the 13-bit field to be able to represent all possible fragment values.

We see that if you divide the maximum possible IP datagram size value – 2^{16} (actually $2^{16} - 1$) by the maximum fragment offset size 2^{13} (actually $2^{13} - 1$) - you have 2^3 which is 8: More simply, 8192 * 8 = 65536. This is how we arrive at the multiplication factor of 8 for the fragment offset length. So, whenever you find a value in the fragment offset, it must be multiplied by 8 to convert to actual bytes that represent the number of bytes of transport header and payload data that came before this fragment in the associated fragment train. What this means too, is that the minimum size for any fragment is 8 bytes and every fragment size is a multiple of 8 bytes except the last fragment.

As an aside, you may see the values of 65535 and 65536 associated with the length and size of an IP datagram. Let's try to clarify this. As mentioned previously, there are 16 bits allocated for the length. The number of discrete values associated with the field is 65536 and those values range from 0 to 65535. Therefore, the maximum possible value is 65535.

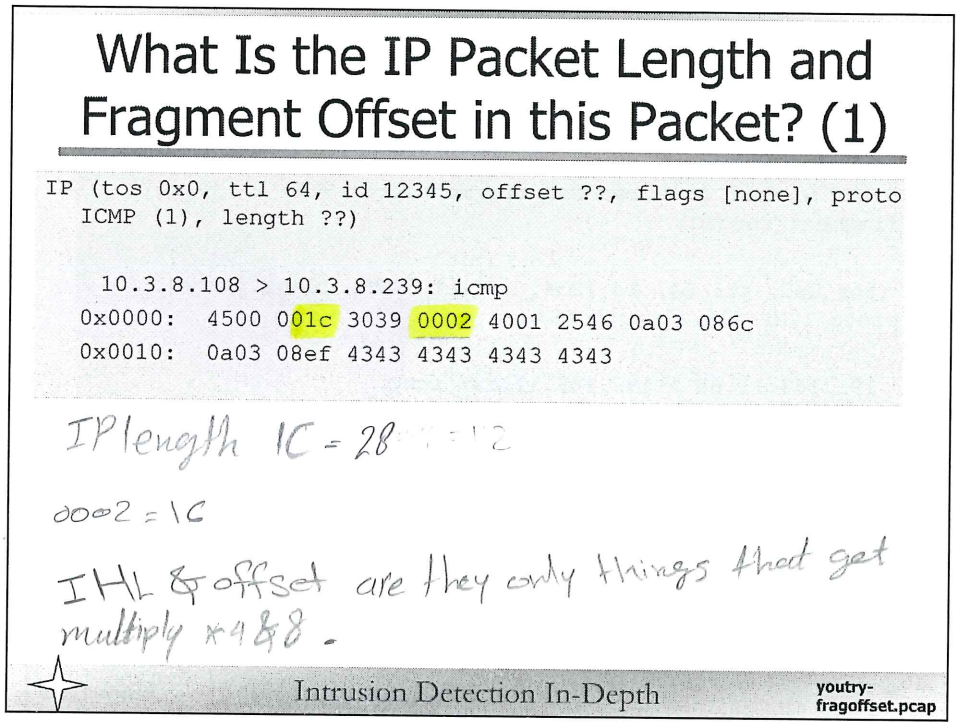
If the use of a 13-bit field to represent a 16-bit value seems convoluted to you, don't worry – it is! Again, this was a bit-saving measure long ago when the use of extra bits was more of a burden.



This record represents the 2nd fragment in the fragment train.

We find a fragment offset of 0xb9 which translates to a decimal 185. But, this must be multiplied by 8 to compute the actual offset which is 1480. This indicates that a previous conventional fragment had a payload of 1480 and an IP header of 20 bytes traveled through an Ethernet network with a MTU of 1500. All fragments should have the same size except the last. That's why we know that a single fragment preceded this one if conventionally generated. It is very possible for someone to craft different sized fragments – typically a sign of abnormal behavior.

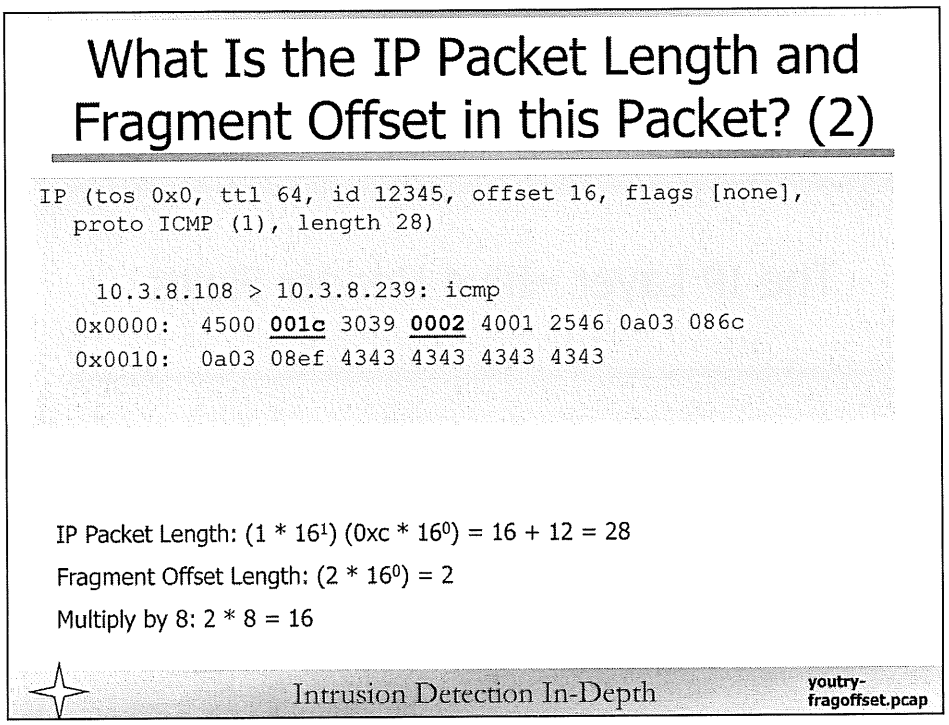
To view this **tcpdump –r fragment.pcap –nt**
output, enter at the command line:
This is output from the second record/fragment only.



Look at the hex dump and figure out the length of the packet and the fragment offset of the second fragment in the pcap.

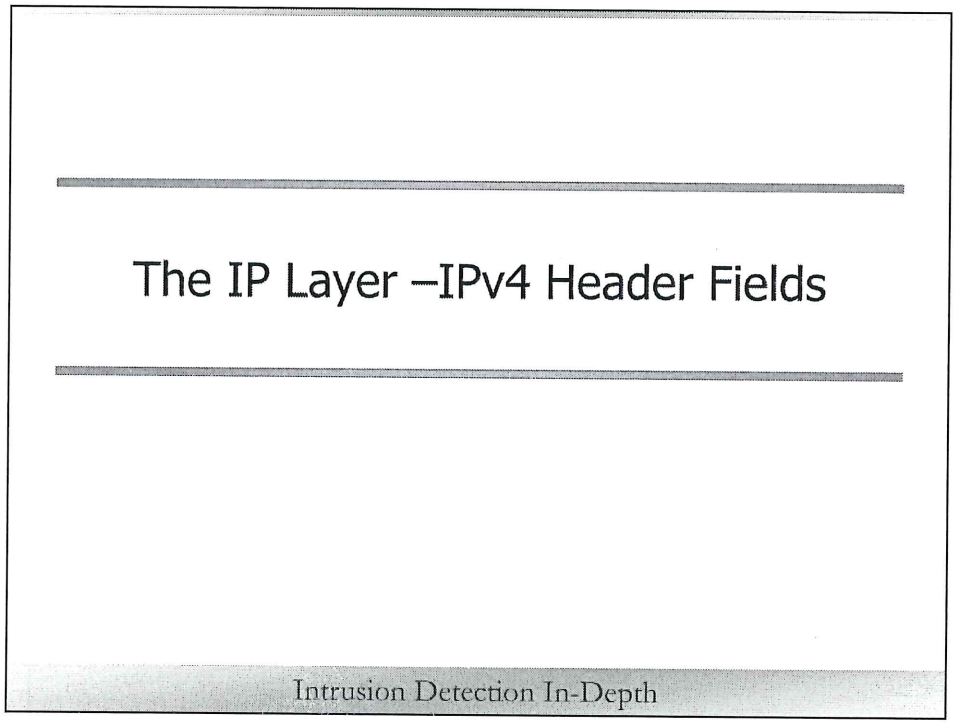
To view the output, enter on the command line:

tcpdump –r youtry-fragoffset.pcap –ntxv

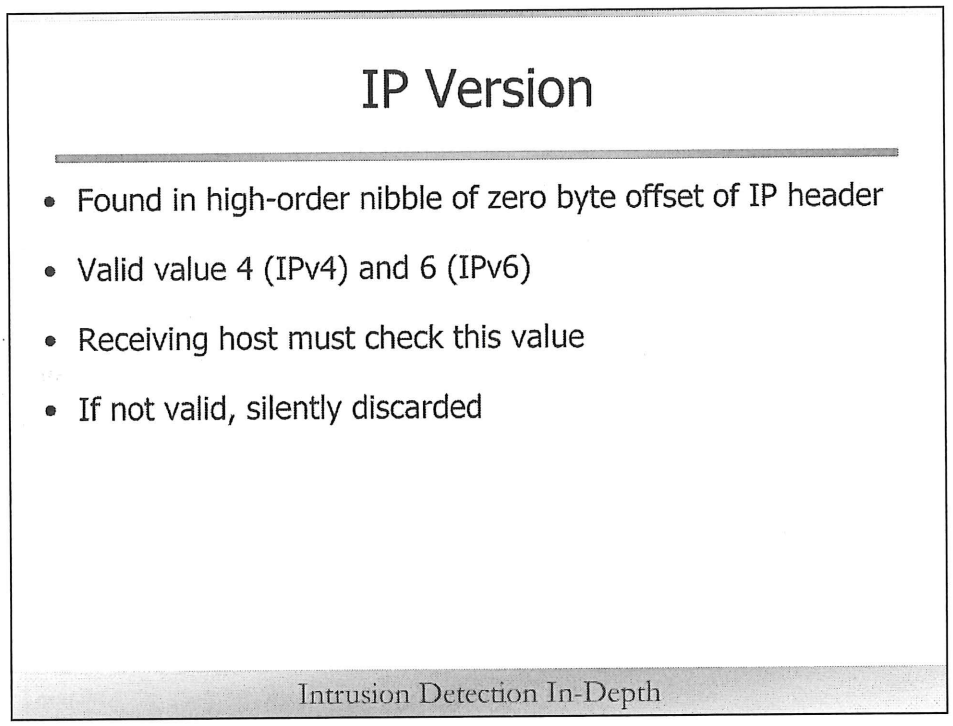


We see that the IP packet length is 28 (20 bytes of header and 8 bytes of data) and that the fragment offset is 16 bytes after the first fragment.

To view the output, enter on the command line:
tcpdump –r youtry-fragoffset.pcap –ntxv

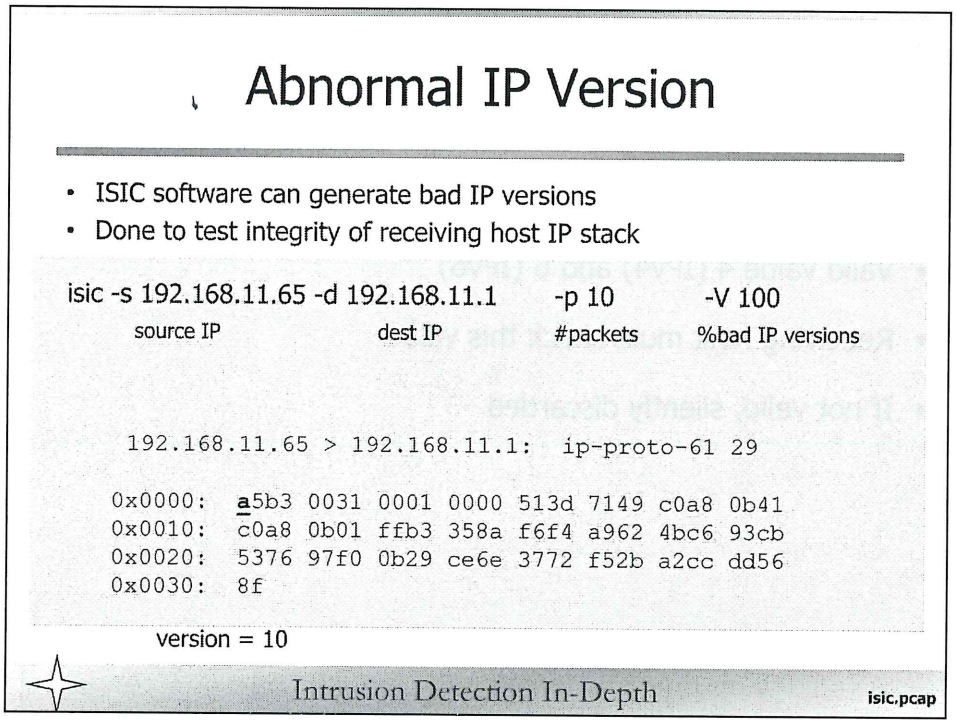


We'll examine the header values in IP in this section. We'll do so with a perspective of the function of a given field as well as in the perspective that an attacker might view the field.



The IP version field must be validated by a receiving host and if not valid, the datagram will be discarded and no error message will be sent to the sending host. RFC 1121 states that the datagram must be silently discarded if an invalid value is discovered. So, crafting a datagram with an invalid IP version would serve no purpose other than to test if the receiving host complies with the RFC. If a packet arrives at a router with an invalid IP version, it should be discarded silently as well.

The only valid version numbers currently in use are 4 and 6, for IPv4 and IPv6, respectively. We cover IPv4 in this section and IPv6 in the next. Perhaps you are wondering what happened to IPv5. It was reserved for an experimental protocol Internet Stream Protocol that later morphed into ST and ST+ that has never been implemented.



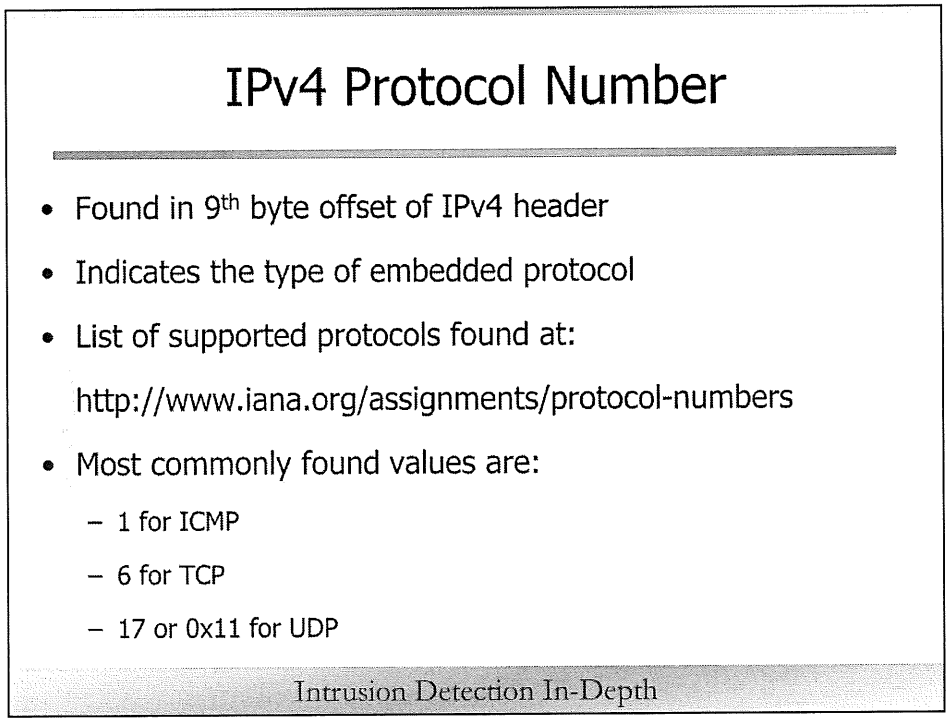
The IP Stack Integrity Checker, ISIC, software is intended to test the integrity of a receiving host's IP stack. It can also be used to see how firewalls or intrusion detection/protection systems react to mutant packets. We generate an isic command to craft abnormal IP version values. We specify a source IP of 192.168.11.65 using the –s option and specify a target host of 192.168.11.1.1 using the –d option. We can designate the number of packets to be sent using the –p option – in this case 10. And, using the –V option we can indicate the number of packets that will have bad IP versions – in this case, all of the packets.

We have captured the output sent using tcpdump. The standard tcpdump output indicates an IP version of 10 (IP10). If you dump the packet in hex, it is easier to see that a bogus version of 0xa or decimal 10 has been generated.

More information on ISIC, including downloads can be found at: http://isic.sourceforge.net.

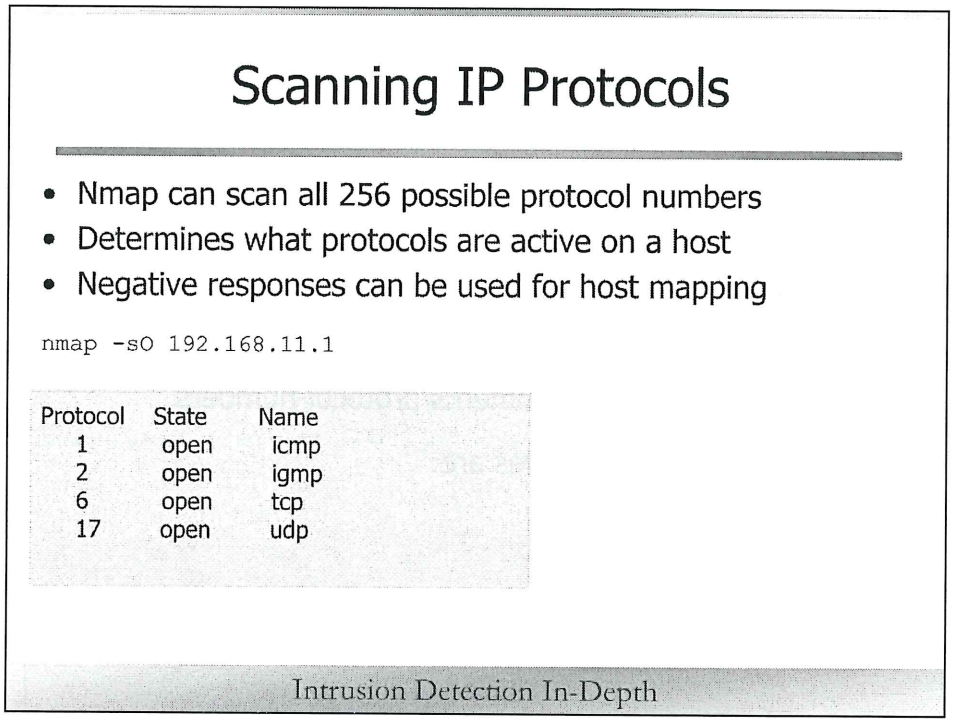
To view the above output, enter at the command line:

tcpdump –r isic.pcap -ntx



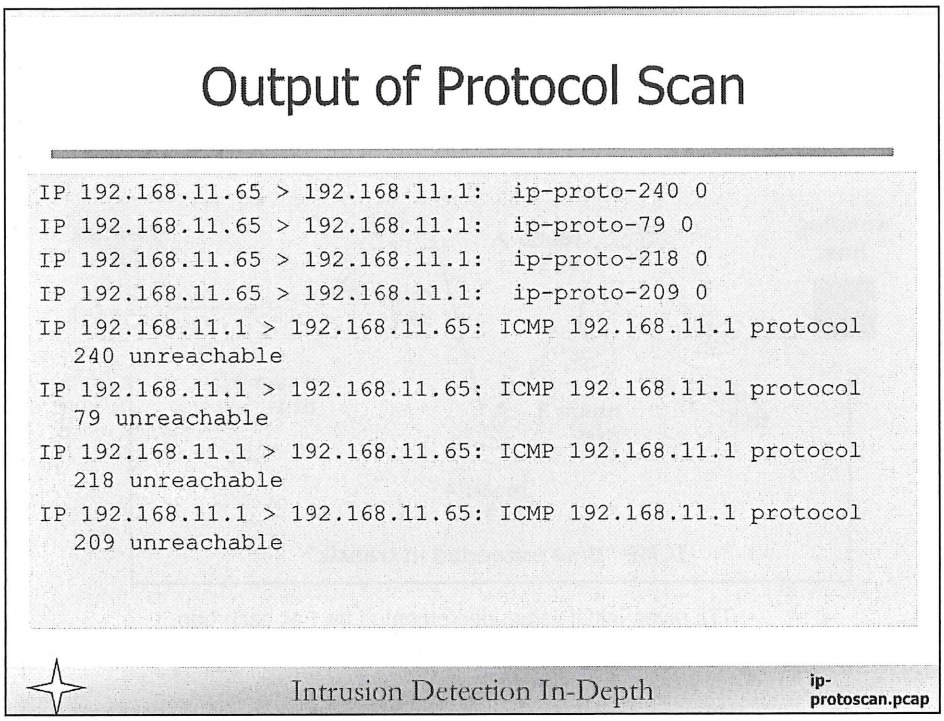
The protocol number found at the 9th byte offset in the IPv4 header specifies the protocol/transport layer that follows the IP header. The most common values for the protocol are 1 for ICMP, 6 for TCP, and 17 or 0x11 for UDP. As we learned early on, one of the pieces of data a given layer needs to know is the protocol that follows it so that it can hand it off to the appropriate software decoder.

If you are decoding a packet displayed in hexadecimal output, this is one of the first fields that you need to examine. That is because you need to know what the subsequent layer is that follows the IP header so that you can interpret it properly.



Nmap is an excellent open-source scanner that we will discuss several times in this course. Nmap is probably better known for its functionality of scanning for open ports on a target host or doing remote operating system identification.

Conveniently, versions of Nmap have the ability to scan a remote host for supported protocols. This is done using the –sO option. The target host is scanned for all 256 possibilities of protocols. Protocols are deemed listening or filtered when no ICMP error message is returned to say that the protocol is unreachable.



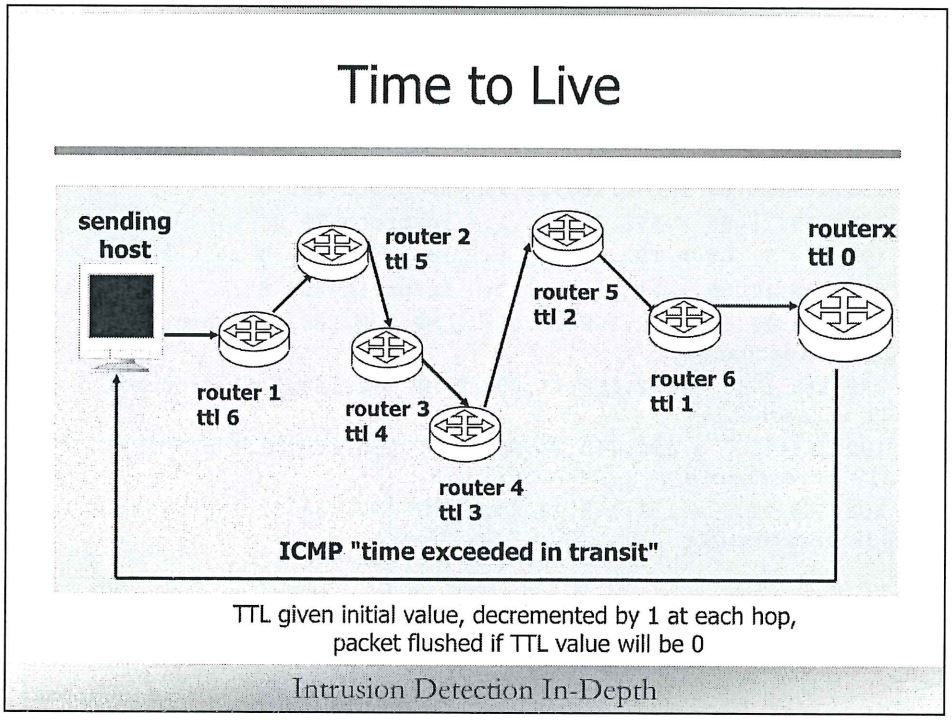
Nmap scans all 256 different protocol types. A host that receives this type of scan should respond with a protocol unreachable message to any protocols that it doesn't support. You can see that 192.168.11.1 is the target IP address and it responds with several different protocol unreachable messages to the scan by 192.168.11.65.

While the supported protocols of a host are mildly interesting, another possible piece of reconnaissance from this type of scan is that the host is alive. This is a more stealthy type of scan. However, if the site has a "no ip unreachables" statement on the outbound interfaces of the gateway router or blocks outbound ICMP, this information will not be leaked to the scanner.

It cannot be assumed that the absence of an ICMP "protocol unreachable" message means that the protocol is listening. Conditions such as the scanned site blocking outbound ICMP messages will prevent the Nmap scanner from getting these messages. There are other conditions such as dropped packets that may also cause the loss of packets and falsely influence Nmap. However, the author of Nmap tried to consider such situations. Nmap will send duplicate packets for each protocol to deal with the problem of packet loss. Also, if Nmap gets no ICMP protocol unreachable messages back at all, it doesn't assume all protocols are listening. Instead, it wisely assumes that the traffic is being "filtered" and reports this.

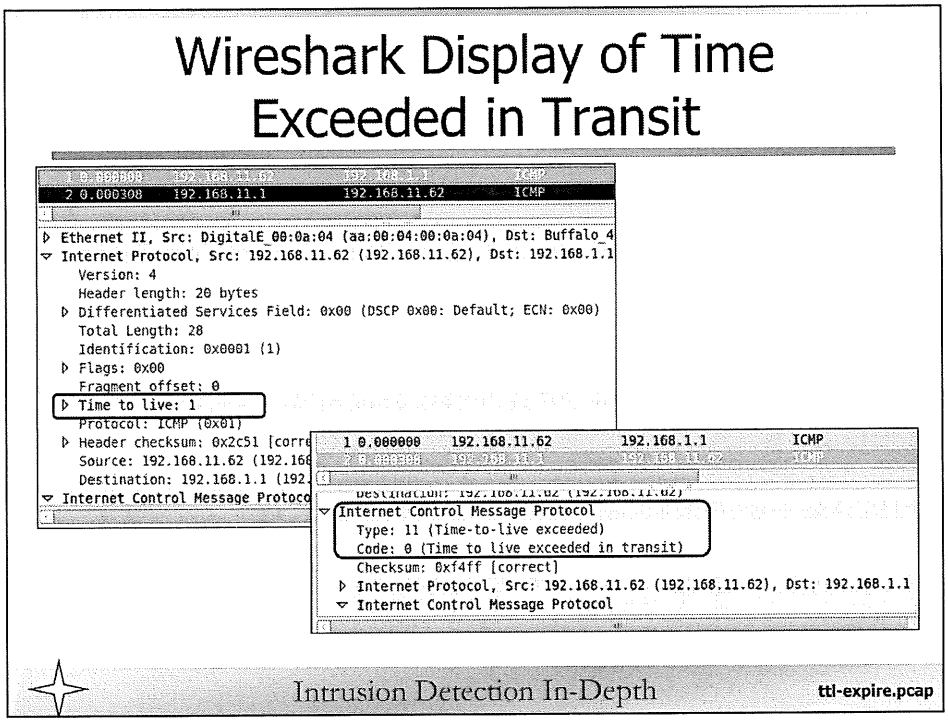
A recurring theme in this course is that the absence of a response – in this case an ICMP "protocol unreachable" message – does not necessarily indicate that a protocol is listening. Let's use a real world analogy to help clarify this concept. Suppose your physician draws some blood on your annual visit and tells you that you will receive a phone call if there are any problems. You cannot assume that the absence of a call definitively means that nothing is wrong. After all, the blood sample may be lost in transit to or from the lab, or the results may be misplaced when returned, or someone may neglect to call you. A similar concept applies to packets.

The output is an excerpt from the following command:
tcpdump –r ip-protoscan.pcap -nt



TCP/IP needs a way to flush a lost packet from the Internet, perhaps a packet that is in some kind of routing loop where it bounces aimlessly among routers. The means used to prevent this wayward packet activity involves a field in the IP header known as the time to live (TTL) value. It's not really a time at all - it is a count of "hops to live" before being discarded.

Different operating systems set different initial TTL values. When a packet traverses a router on its travel from the source to destination, each router will decrement the TTL value by 1. If the value ever becomes 0, the router will discard the packet and send an ICMP "time exceeded in-transit" message back to the sending host.

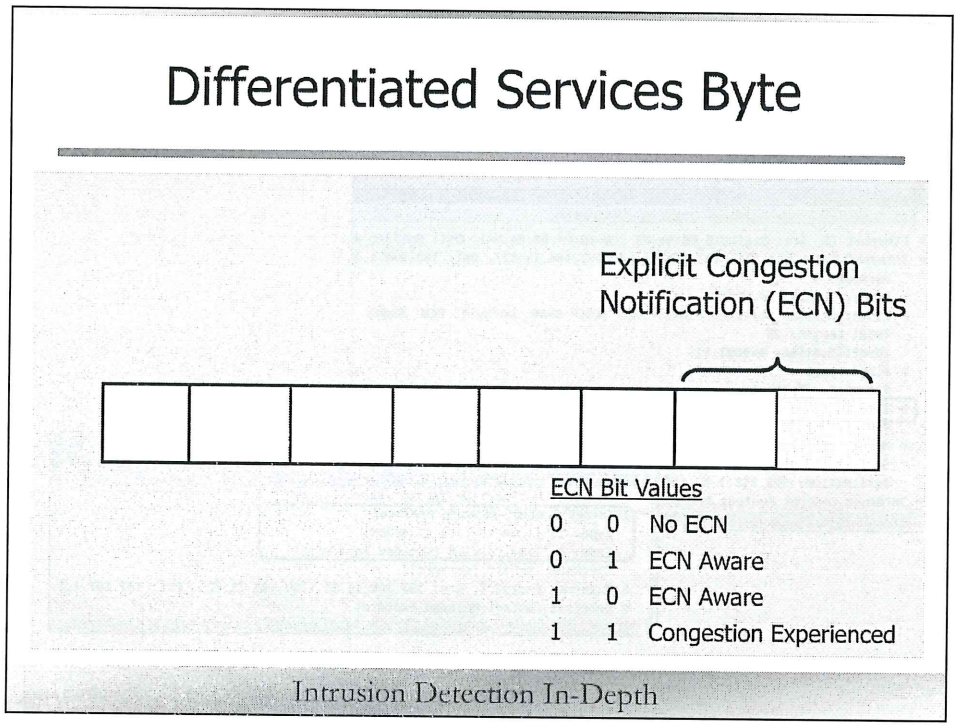


The two different extracts of Wireshark displays illustrate what happens when the TTL expires. The upper output shows a host that sends an ICMP echo request that needs to traverse two routers to get to 192.168.1.1, however the TTL has a value of 1.

The lower display shows the results. We haven't covered the specifics of ICMP just yet, but you can get the general idea of what has transpired. The router returns an ICMP "Time to Live exceeded in transit" message, indicating that the packet cannot be forwarded. You see an IP and ICMP layer depicted beneath Wireshark's interpretation of the ICMP error message. When we examine ICMP more thoroughly, we'll discuss how an ICMP error message contains part of, or the entire sender's packet in the limited number of bytes allocated for that purpose. This informs the original sender's TCP/IP stack which sent packet caused the error.

To view the output, enter on the command line:

wireshark ttl-expire.pcap

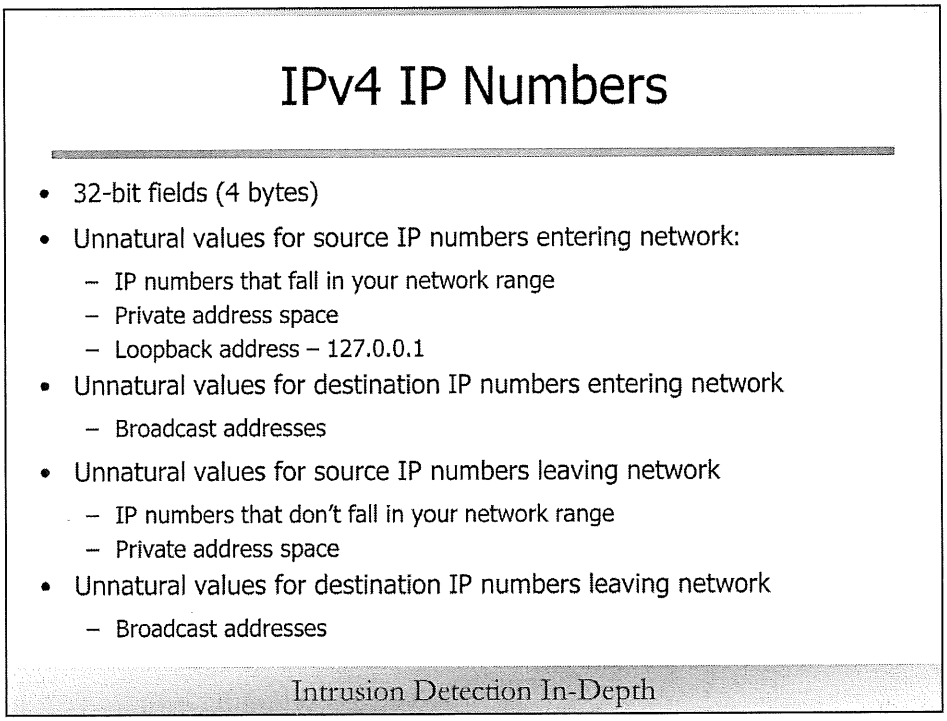


A field originally known as the Type of Service byte, found in byte 1, now known as the differentiated services field, has undergone several rounds of alterations since its incipient specification. One of these alterations in RFC 2481, and more currently RFC 3168, calls for the two low-order bits of the differentiated services field byte to be used for Explicit Congestion Notification. The purpose here is that some routers are equipped to do Random Early Detection (RED) or active queue management of the possibility of packet loss.

When congestion is severe, it is possible that a router can drop packets. RED attempts to mitigate this condition by calculating the possibility of congestion in the queue to a router interface and marking packets that might otherwise be dropped as experiencing congestion. If the ECN-aware bits are set in the differentiated services field, that indicates the sender and receiver are ECN-aware. If the end-hosts are ECN-aware, the router will attempt not to drop the packet, but instead send it with the Congestion Experienced (CE) bits enabled and the receiver will respond appropriately. We'll discuss the receiver's response in more detail when we cover the TCP fields in the next section. Currently, this mechanism is available only for TCP. ECN has really failed to catch on, yet operating systems continue to support it. When we discuss TCP, we'll discover that there is a TCP component associated with ECN – namely two TCP flags that assist in identifying that state of ECN – that work in conjunction with the IP header differentiated services byte.

ECN Bit Values

0	0	Either one or both hosts are not ECN-aware or they are in the process of negotiating if they are ECN-capable (done during SYN and SYN-ACK exchange of three-way handshake)
0	1	Either of these next two pairs of bit settings indicates that both end-hosts are ECN aware after the SYN and SYN-ACK exchange
1	0	
1	1	Congestion has been experienced and been marked by the router

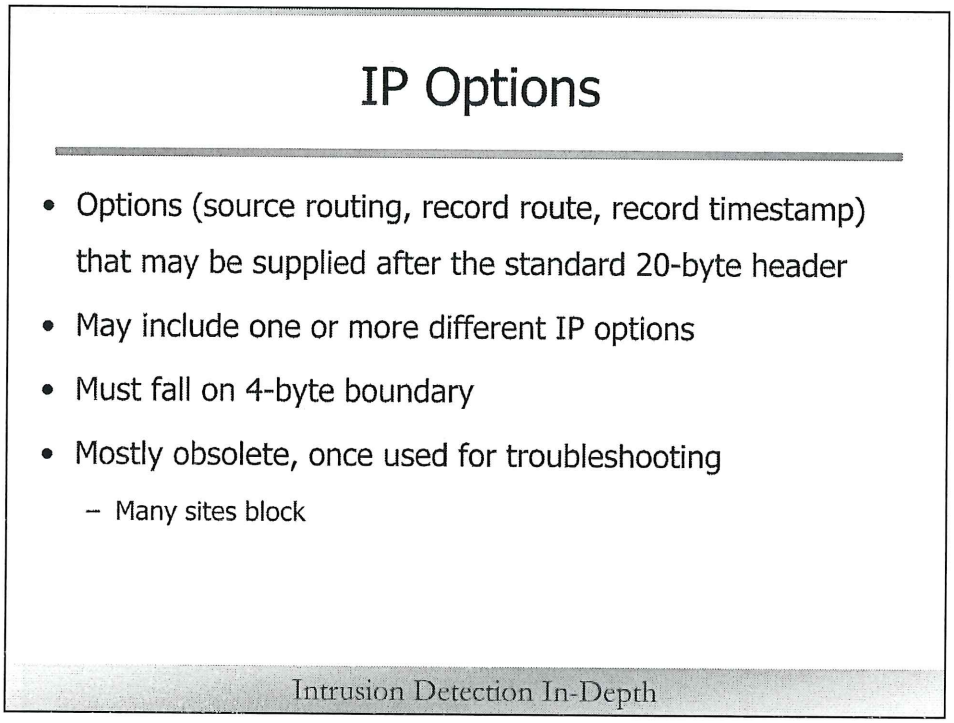


The source IP number is located in the 12th-15th bytes offset of the IPv4 header; the destination IP number is located in the 16th-19th bytes offset of the IPv4 header. If you see an IP number entering your network that purports to be from your network, there is a problem. Most likely someone has crafted this packet. A packet-filtering device should shun this traffic.

Traffic leaving your network should have a source IP number that reflects your network's address space. If you see an IP number that originates from inside your network, yet has an IP number of a different address space, it is either being spoofed or there is a misconfiguration problem with a host. In either case, this traffic should not be allowed to leave your network. This will prevent hosts in your network from participating in distributed denial of service attacks since participant hosts usually use spoofed source IP number so that they cannot be located. Other types of scans use "decoy" or spoofed source IP's as a smokescreen. By disallowing outbound traffic that is not part of your address space, these will be ineffective as well.

You should also never see the loopback mode address leaving or entering your network since that identifies the local host. Internet Assigned Numbers Authority (IANA) reserved private network addresses such as 192.168.0.0/16 and 172.16.0.0/12 are intended to be used for local networks only and are not supposed to be routed outside of your network. These address ranges can be found at:

http://www.iana.org/assignments/ipv4-address-space

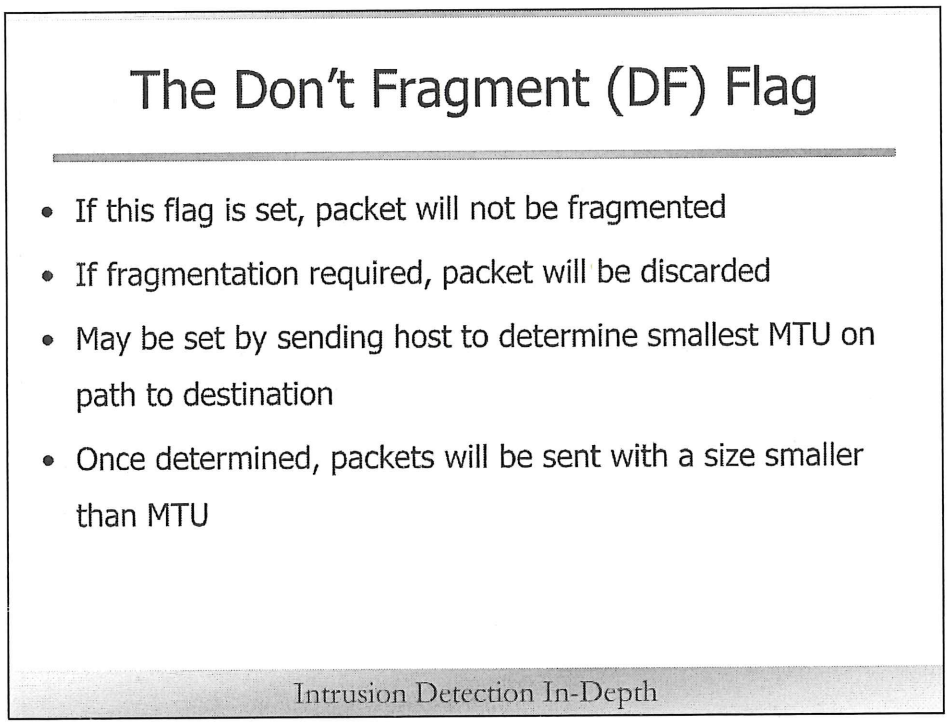


We briefly discussed IP options – mostly in terms of their effect on the IP header length. IP options, as the name suggests, are optional parameters that can be placed in, at maximum, 40 bytes after the standard IP header. Mostly, though, these are obsolete and tend not to be used – or perhaps that should be rephrased to tend not to be used for benevolent purposes. Many sites block packets that have IP options.

Where space permits, there may be multiple IP options following the IP header. Each is distinguished by an IP option type, a length value if the IP option is greater than one byte, and the accompanying value. The length field is used to find the end of the current IP option and the beginning of the next, if one follows. The total field of aggregate IP options must fall on 4-byte boundaries. If they do not naturally end on a 4-byte boundary, they are padded with either NOP (0x01) or EOL (0x00) bytes to extend the field to a 4-byte boundary.

You especially should be aware of the danger of allowing IP options loose routing or strict routing. They specify the routers through which the packet should travel on its way to the destination. Strict source routing requires all the routers along the route to be designated in the IP options; while loose routing requires you to designate some of the routers and allow normal routing to find others that might be necessary to send the packet to the destination. These options were intended to be used for troubleshooting. When the destination host responds, convention dictates that the same routing data be placed in the IP header, but in reverse order.

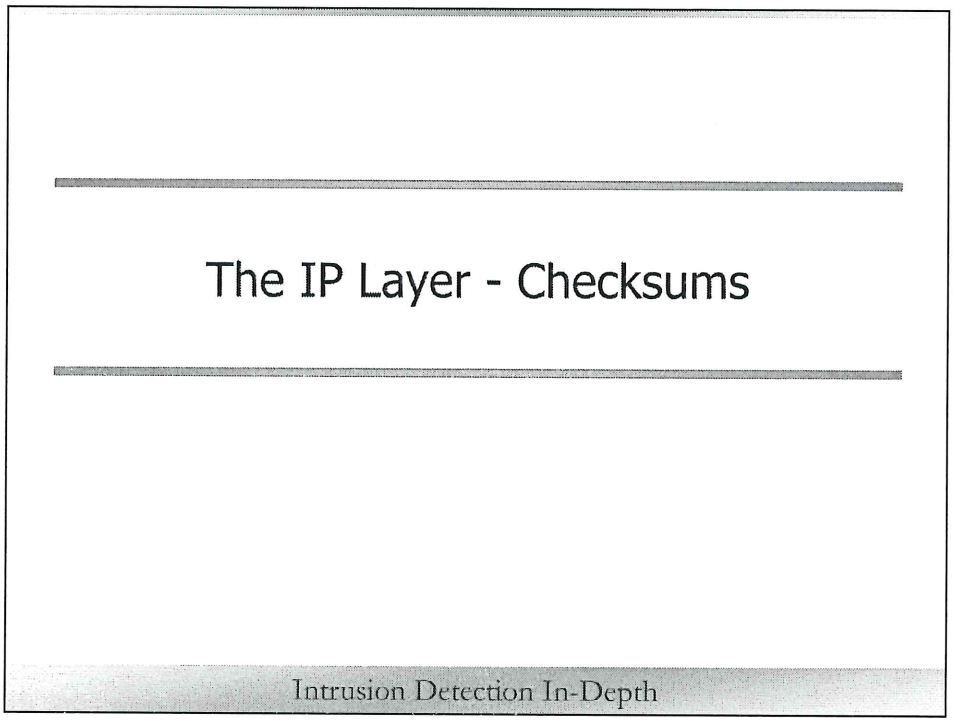
Suppose an attacker is able to route traffic through a node under her/his control. This does not bode well if an attacker can spoof a source IP and is able to redirect and possibly alter traffic to the node under control.



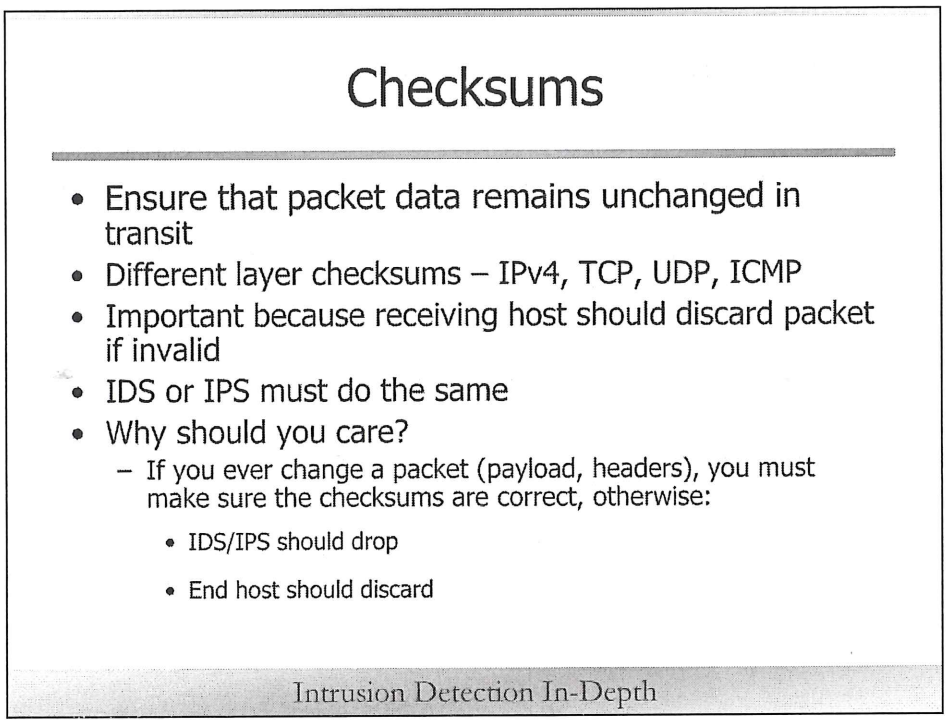
As the name implies, if the Don't Fragment flag is set, the packet will not be fragmented by any device through which it passes. If this flag is set and the packet crosses a network where fragmentation is required, the router will discover this, discard the packet and send an ICMP error message back to the sending host.

The ICMP error message will contain the MTU of the network that required the fragmentation. Some hosts intentionally send an initial packet across the network with the DF flag set as a way to discover the MTU for a particular source to destination path. If the ICMP error message is returned with the smaller MTU, the host will then package all packets bound for that destination into small enough units to avoid fragmentation. This process is known as path MTU discovery.

Fragmentation comes with some overhead so it is desirable to avoid it altogether. As you will learn in the fragmentation section, if one fragment is not delivered, all fragments will have to be re-sent. Because of this, when some TCP/IP stacks send data, they will send a discovery packet with the DF flag set. If the packet goes from source to destination without any ICMP errors, then the selected datagram size of the discovery packet is used for subsequent packets. If an ICMP message is returned with an unreachable error – "need to frag" message and the MTU is included, then the packet is resized so that fragmentation does not occur. This assumes the site allows these ICMP messages inbound to receive the MTU information.



This page intentionally left blank.



Checksums are more complex in theory than many of the other IP header fields so we'll digress to discuss them. Checksums are a method used to ensure that data has not gotten corrupted in transit from source to destination. There is a standard algorithm used to compute checksums. The sending host applies this algorithm to all the fields involved in the checksum computation and places the resulting value in the appropriate header's checksum field. For instance, the IPv4 header checksum value is computed by applying the checksum algorithm to all fields in the IP header and the resulting value is placed in the IP checksum field – a 16-bit field in the IP header.

The receiving host must validate that the checksum is correct. It does so by applying the same algorithm over the fields in the IP header. If the resulting value matches the value in the IP checksum field in the IP header, the packet is passed up to the transport layer. If the value does not match, the packet is silently discarded.

The algorithm used for TCP/IP is to divide the data that is being checksummed into 16-bit fields. Each 16-bit field has a 1's complement operation done on it and each of these 1's complements values are added. The final value is considered to be the checksum. A 1's complement is an operation that flips/inverts all the bits in a value.

As we'll see, different layers of the TCP/IP stack have their own checksums. For instance, the IP checksum is a value that routers forwarding packets must validate after decrementing the TTL. The transport layers have their own checksums to make sure that header and data information was not changed in transit.

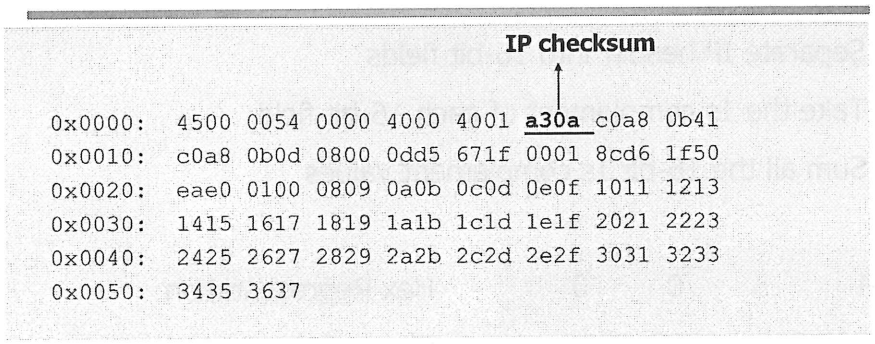
Why do we really care about checksums? They are an important consideration when interpreting or crafting packets. They can also become an issue if you are trying to understand why your IDS or IPS may be discarding packets that seem perfectly reasonable to you. Destination hosts or devices must validate checksums and discard packets with invalid ones. An IDS or IPS must discard invalid packets with invalid checksums. If you are using an IDS/IPS such as Snort and you think you should be getting alerts on some traffic you are testing,

but you see no alerts – one of the first things you should examine is that your packets have correct checksums on all layers – IP, TCP, UDP. Otherwise, they will be dropped or discarded by Snort. If you check the mailing list archives for Snort, you'll see that this topic has been discussed several times.

Also, if you craft or alter packets that you send to an end host or run past an IDS/IPS, you must make sure that the program that you are using recomputes the checksums.

Note: RFC 1071 "Computing the Internet Checksum" uses the terminology of "1's complement" when performing a part of the checksum computation. In actuality, the computation involves something known as "2's complement". Don't worry about the difference in the two. This is mentioned only because the terminology used in this course is "1's complement" to agree with the RFC.

IP Checksum



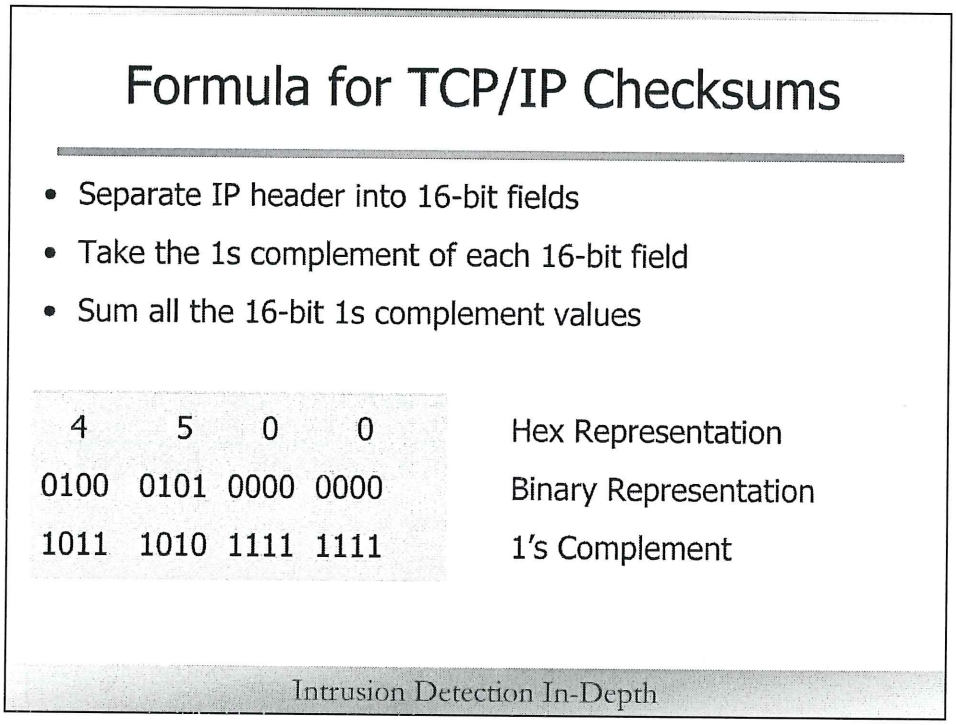
Intrusion Detection In-Depth

The IP checksum is found in the 10th and 11th bytes offset of the IPv4 header. This IP checksum covers all fields in the IP header only. This checksum is different than the checksums that are computed for the embedded protocol fields because it is validated along the path from source to destination. Embedded protocol checksums such as TCP, UDP, and ICMP are validated by the destination host only. The IP checksum is validated by each router through which it passes from source to destination and finally validated by the destination host as well.

If the computed checksum does not agree with the one found in the datagram, the datagram is discarded silently. No attempt is made to inform the source host of a problem. The idea is that higher level protocols or applications will detect this and deal with it.

In IPv6, IP header checksums are eliminated altogether. This becomes more efficient since any device such as a router that alters the TTL value must recompute the IP checksum. IPv6 requires the transport layer only to carry a checksum.

If IP version is wrong or Check sum is bad
The packet should be dropped silently



The formula above is shown for the IP header checksum, but it is used for all other IP datagram checksums as well. In the case of the IP header, we divide it into 16-bit fields. Since the IP header length is always a multiple of 4 bytes, we will not have to worry about extra fields that do not fall on 16-bit boundaries.

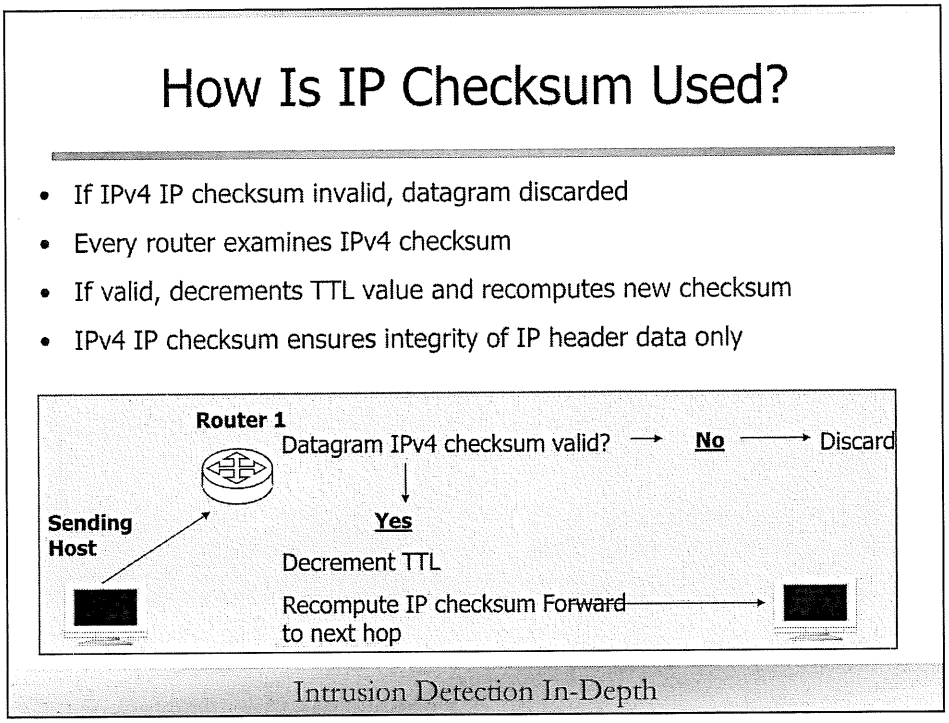
Once all of the fields are separated, we take the 1s complement of each. This operation simply flips the bit. All of these individual 1s complement values are added to form the checksum.

Above you see the first 16 bits of a very common beginning to a datagram. Each hex value is represented in 4 binary bits and each of these bits is flipped. This becomes the 1s complement value. This operation is commutative so you can add the hex values of the 16-bit fields and then take the 1s complement and the resulting checksum should be the same.

We show all pairs of 16-bit fields and represent the 1's complement values and add them together. This is a fairly straightforward operation, but there is a consideration we must note. When the high-order bits of both 16-bit fields have a value of 1, we get a result of 1, but we also need to carry over a 1. This cannot be discarded; it must be wrapped around and added to the low-order bit. The actual addition of the carry-bit may be deferred so that it is optimized by adding them all at once to the low order bits. This is possible because the operations applied in the checksum are commutative – the order in which they are performed does not matter.

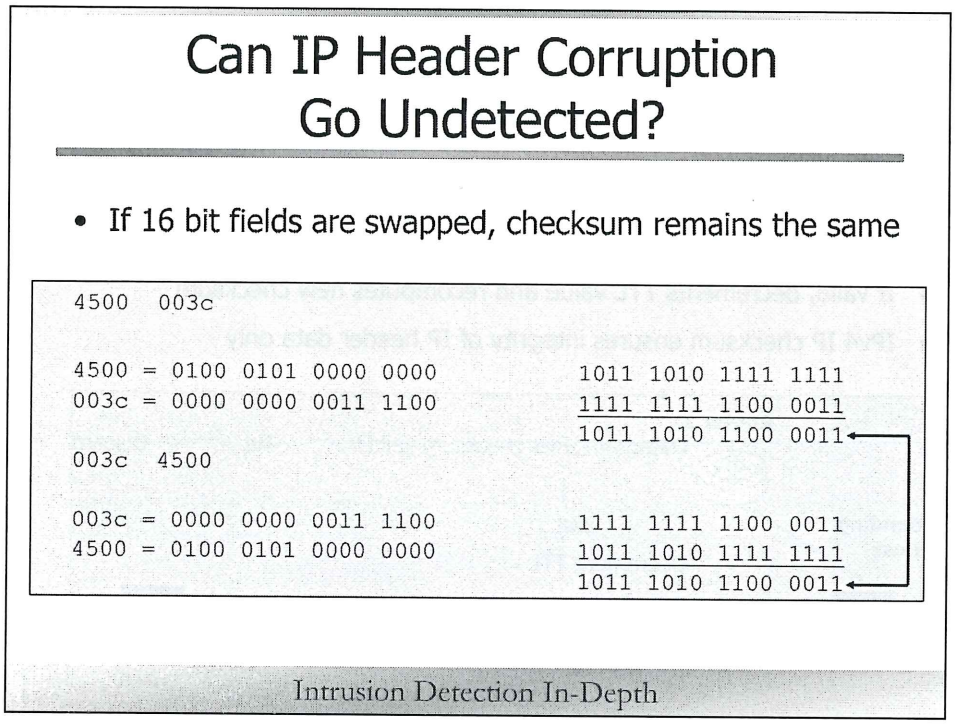
This same process is performed for all the 16-bit fields that are eventually summed together to derive the IP checksum.

To read more about IP checksums, look at RFC 1071.



The IPv4 IP checksum is examined and recomputed for each hop on the way from source to destination. Every router that examines the datagram must validate the checksum. If it is invalid, the datagram is silently discarded. If it is valid, the TTL value is decremented by 1. This will change the value of the actual IP checksum so it needs to be recomputed, placed in the IP header field, and then sent on its way. Remember this checksum validates the fields in the IP header only – not the rest of the datagram that consists of the embedded protocol header and data.

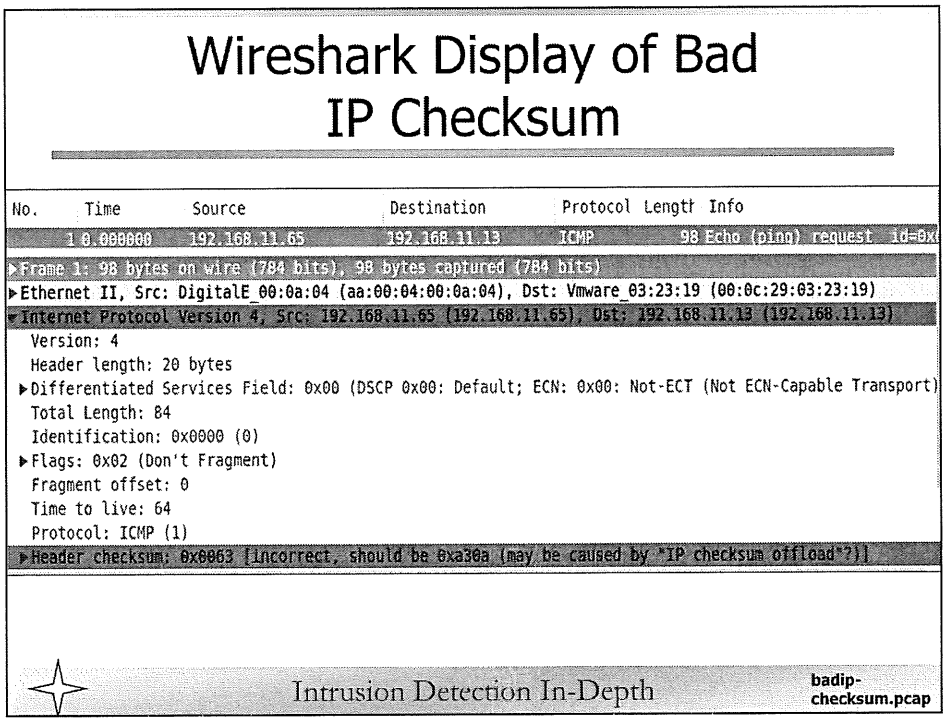
The rationale for checking the IP checksum for each hop makes sense when you think about it. The worst case scenario is that the destination IP becomes corrupted. It makes no sense to forward a packet that has been corrupted because the corruption may alter a field like the destination IP address.



While the IPv4 IP checksum and all other embedded protocol checksums found in the datagram will find most packet corruption, there is a problem. It is possible for entire 16-bit fields to be swapped and yet the checksum will remain the same.

As you can see, we try this for the first 2 16-bit fields in the IPv4 header. The computed checksum for the first 2 fields is 1011 1010 1100 0100. But, if we reverse the fields and compute the checksum, it is exactly the same. A datagram with 16-bit fields swapped is a vastly different datagram in meaning and resolution when fields are swapped. So, this is a drawback of using this computation.

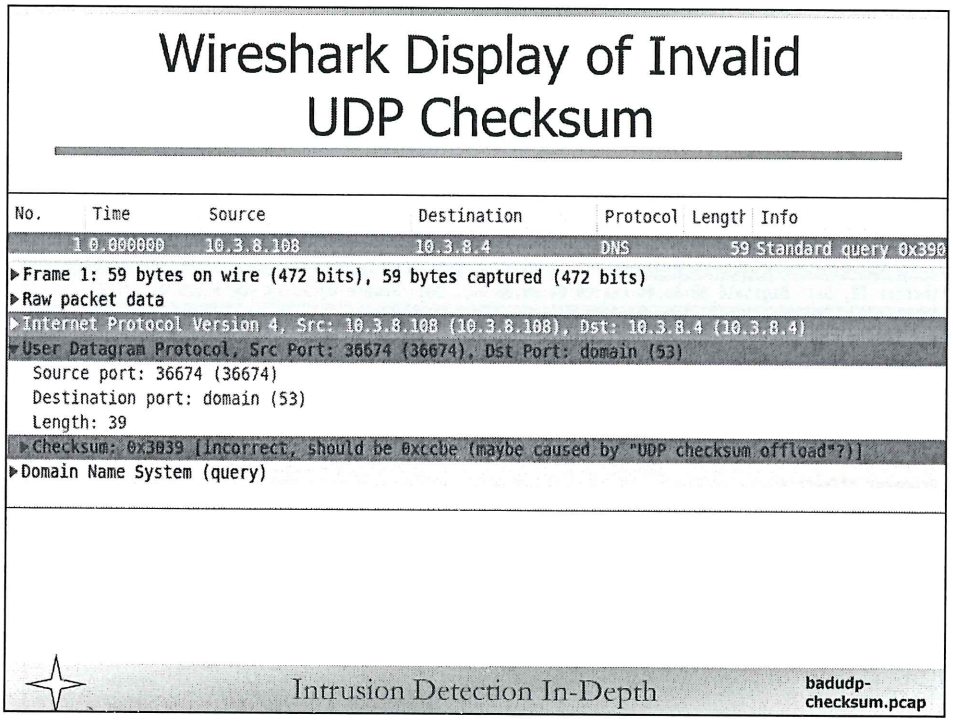
Why not use a more complicated and reliable algorithm for the checksum? This computation is done for each packet that a router or host receives. The simpler the algorithm, the quicker the computation time. The checksum algorithm is a fast and mostly reliable algorithm and the exact swap of 16-bit fields is a rare occurrence.



Wireshark computes IP, TCP, UDP checksums if the preferences are configured to enable checksum validation. It highlights in red any protocol that has an invalid checksum. For instance, there is an obvious invalid IP checksum in the display thanks to Wireshark's highlighting. This displayed packet will be silently dropped at the next node – router or destination host.

To view the output, enter from the command line:

wireshark badip-checksum.pcap



We have not talked about the transport layer yet. But, TCP, UDP and ICMP all have their own checksums. The receiving host validates these checksums and silently drops a packet where the embedded protocol checksum is invalid. For instance, there is an invalid UDP checksum in the display. Wireshark informs us that the checksum is 'incorrect, should be 0xccbe (maybe caused by "UDP" checksum offload?")'

Checksum offload refers to the capability of having the NIC take care of all checksum computations. This frees the protocols from having to perform the computation. While the checksum formula is efficient, in aggregate, it does take CPU resources. The checksum offload is a valid explanation for a bad checksum only for outbound traffic because Wireshark views the traffic before it reaches the NIC. Inbound traffic should have traveled from the NIC where the checksum computation has already occurred by the time Wireshark sees the packet.

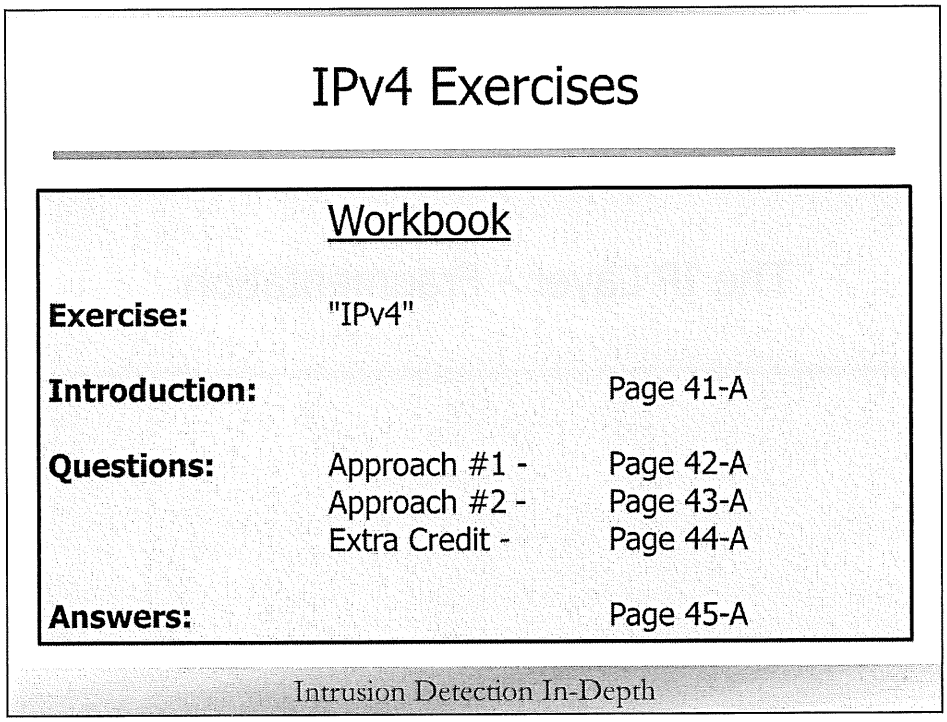
Wireshark must be configured to validate UDP checksums. To do so, do the following:

Edit → Preferences → Protocols (bottom of the first column)

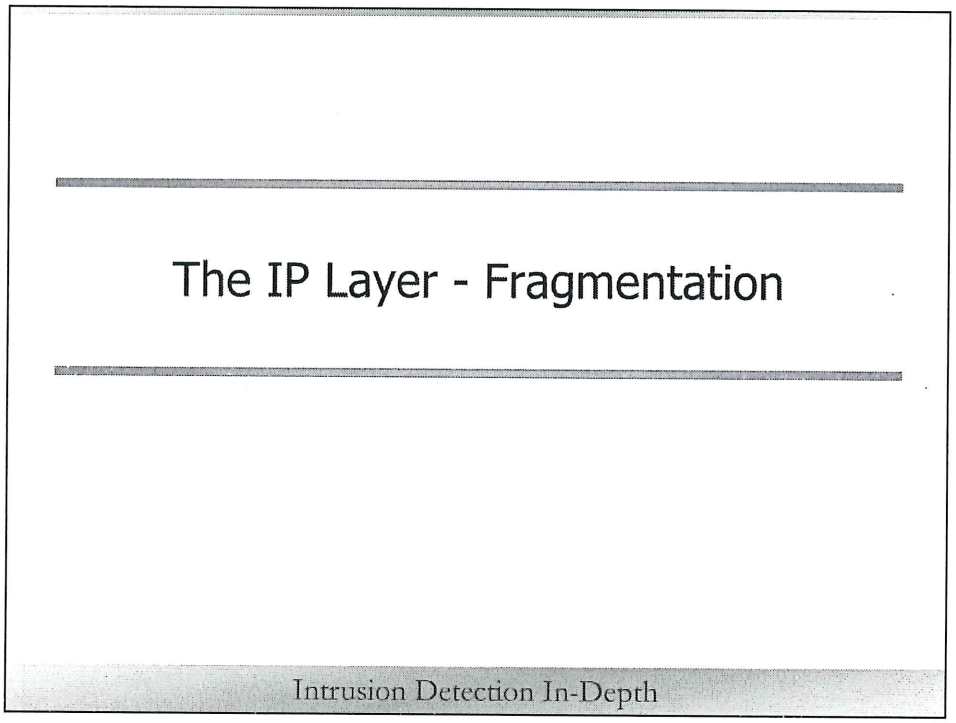
Expand Protocols and scroll down to UDP. Check "Validate the UDP checksum if possible:"

To view the output, enter from the command line:

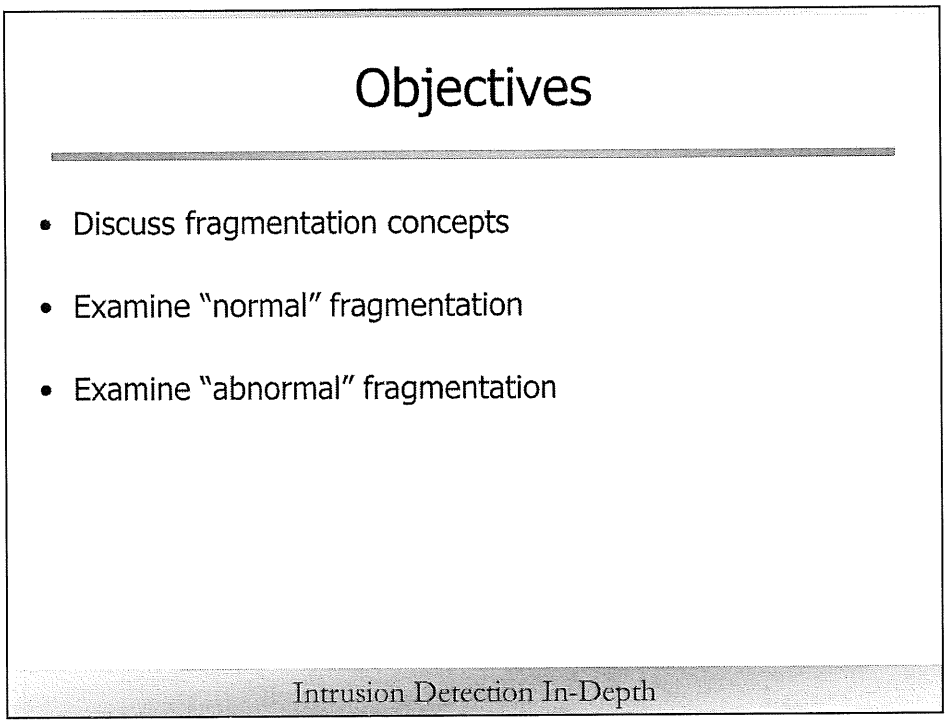
wireshark badudp-checksum.pcap



This page intentionally left blank.



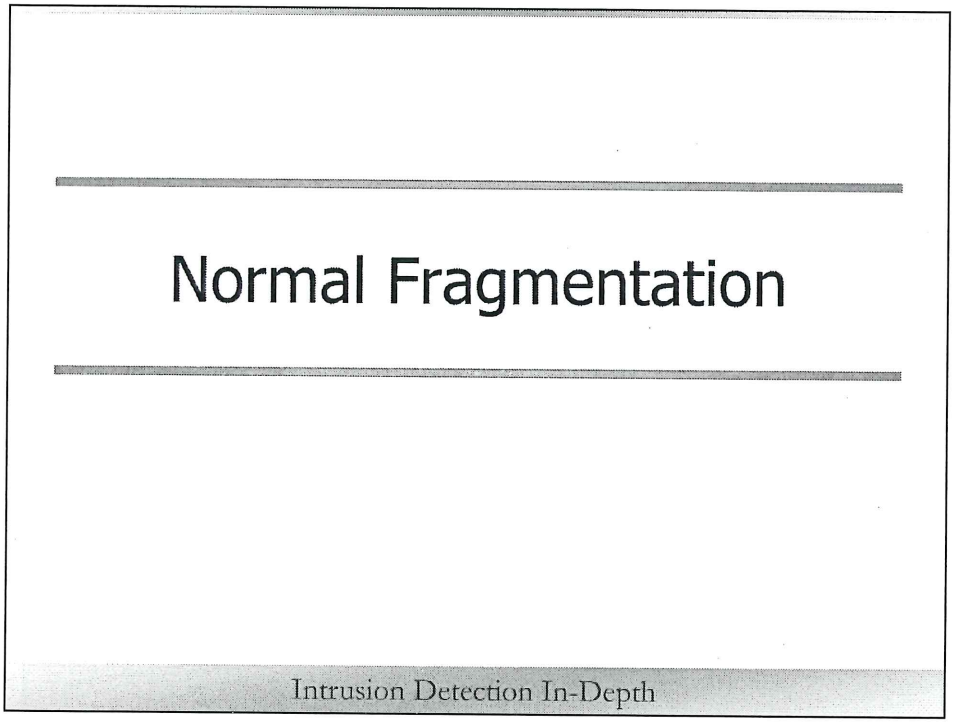
This page intentionally left blank.



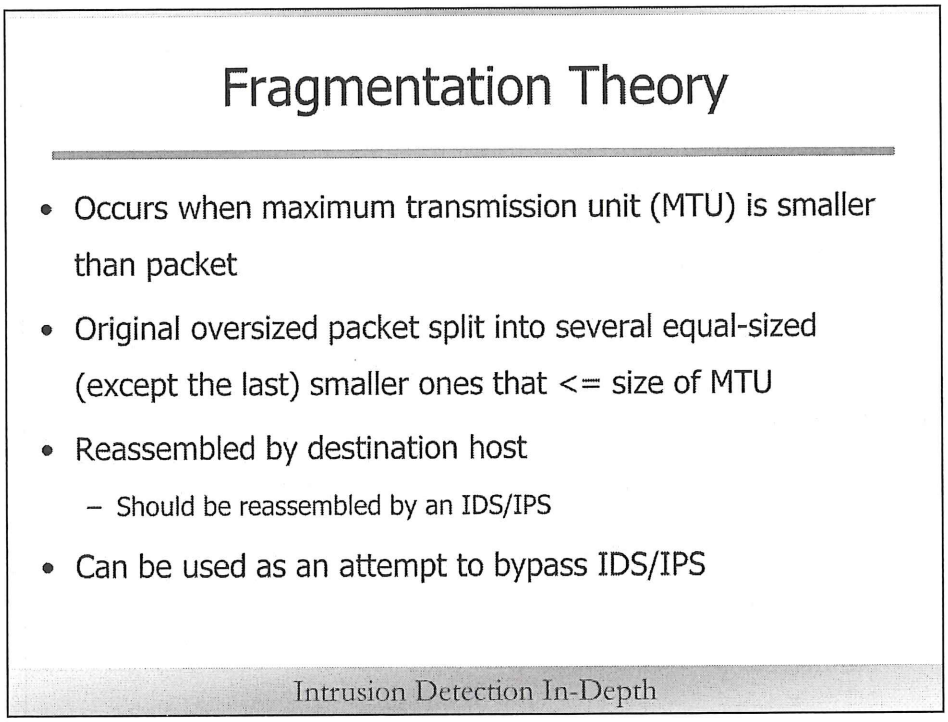
Attackers can attempt to use fragmentation to mask their probes and exploits. There are availability (or denial of service) attacks that use highly fragmented traffic to exhaust system resources. These are some of the many reasons that you may want to learn about fragmentation.

By understanding how this facet of IP works, you will be equipped to detect and analyze fragmented traffic and discover if the fragmentation you encounter is normal or if it's being used for more nefarious purposes.

We will look at fragmentation to see what is happening at the packet level. We need to be aware of "normal" fragmentation before we can identify "abnormal."



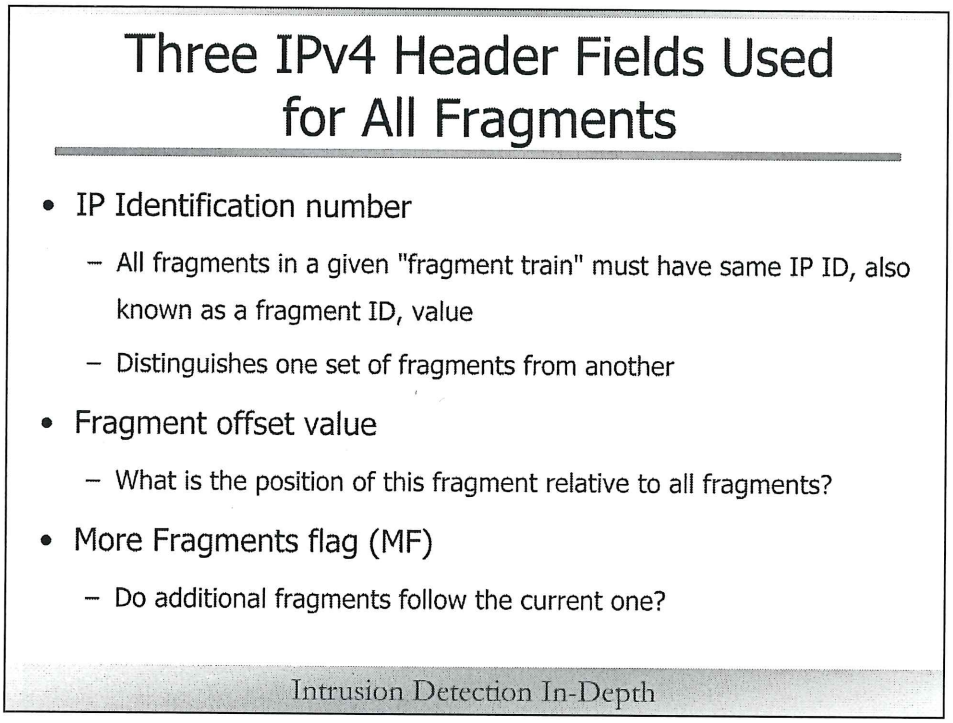
Fragmentation can be a very normal and naturally occurring effect of traffic travelling between variously sized networks. We will consider the theory and composition of normal fragmentation first to acquaint you with how it should operate.



Fragmentation occurs when an IP packet travelling on a network has to traverse a network with a maximum transmission unit (MTU) that is smaller than the size of the packet itself. For instance, for Ethernet, the maximum transmission unit or maximum size for an IP packet is 1500 bytes. If an IPv4 packet that is larger than 1500 bytes needs to traverse an Ethernet link it must be fragmented. This may be done by a router or the sending host.

The original oversized packet is split into two or more same sized (except for the last) packets that are less than or equal to the MTU size. The last fragment contains the leftover or remainder of the original packet that probably is not an exact multiple of the MTU size. Each fragment has an IP header that is similar to the original oversized packet, with modifications to each to express unique values associated with each fragment. We'll use the term "fragment train" to mean all related fragments that were created from the original oversized packet.

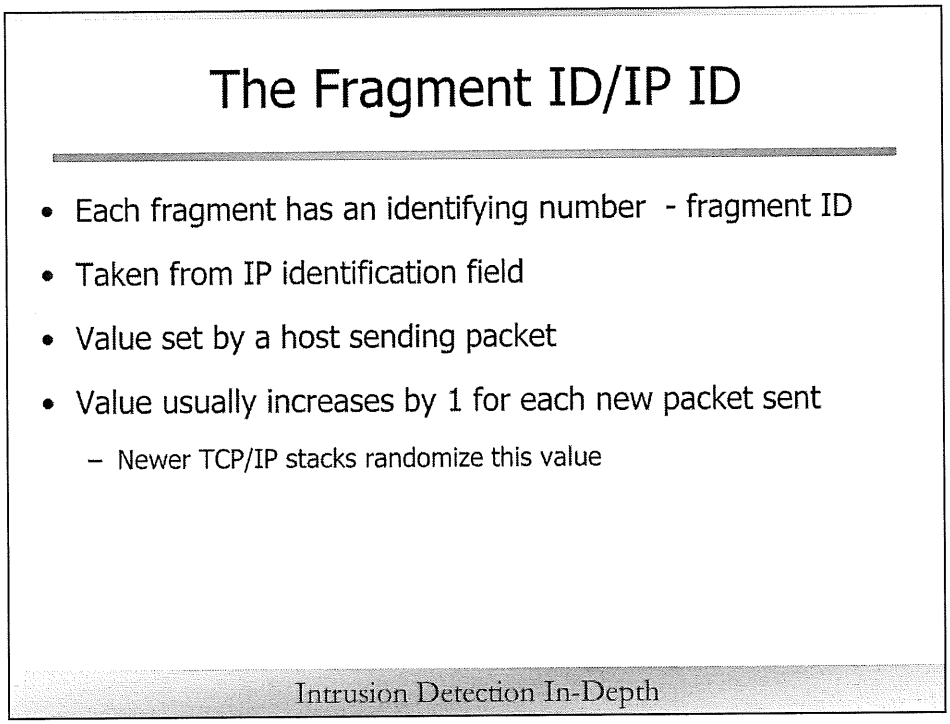
Fragments continue on to their destination where they are reassembled by the destination host. While fragmentation is a perfectly normal and naturally occurring event, it is possible to craft fragments for the purposes of evading detection by IDS/IPS solutions that don't deal well with fragmentation.



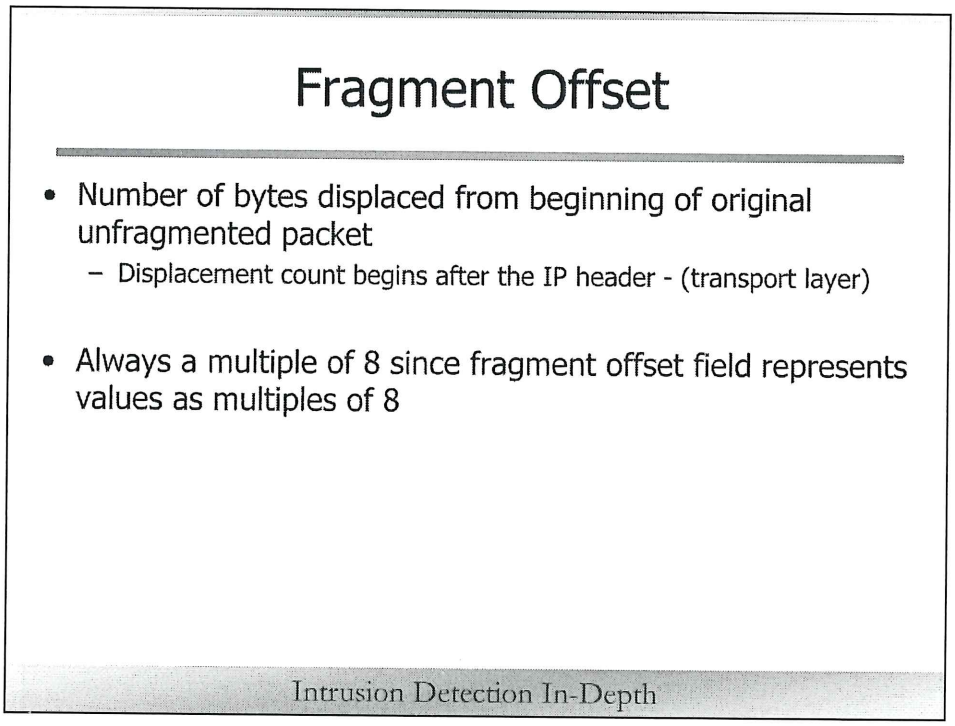
Each fragment has its own IP header in order to get from source to destination. The IP headers of all fragments share the same IP addresses, the same next protocol, and other miscellaneous fields as well. The fields we need to understand for fragmentation are the IP ID, the fragment offset value, and the MF value. The IP ID number is 2-byte number (0-65535), typically uniquely generated when the original packet is first created. If fragmentation occurs, all the resulting fragments (fragment train) share this same IP ID value. If a subsequent oversized packet from the original sender gets fragmented too, it will have a unique IP ID value in all fragments, distinguishing the first set from the second set of fragments. The receiving host knows which fragments are associated with each other by the unique IP ID value.

The portion of the original packet that becomes fragmented is the protocol header following the IP header and the payload following the protocol layer from the original oversized packet. As we learned, packets - and fragments are packets as well - may travel different routes from source to destination and there is no guarantee of arrival nor is there a guarantee that they will arrive in the same order in which they were sent. Therefore, it is imperative for each fragment to have a means to identify its place or chronology among all the other fragments. The fragment offset is used to determine chronology. The fragment offset value is relative to the beginning of all the fragmented data (the protocol header and payload following the IP header). The receiving host uses the fragment offset to order all fragments that share the same IP ID value, source and destination IP's and embedded protocol.

The receiving host has to know whether all fragments have arrived; after all there is no guarantee of delivery. So far, we have a means only to associate fragments via the IP ID number and a method of specifying each fragment's place in the entire fragment train. The one-bit MF flag designates whether or not more fragments follow. A value of 1 means that there are more fragments, a value of 0 means there are no more fragments. This means that every fragment except the last one should have the MF bit set.

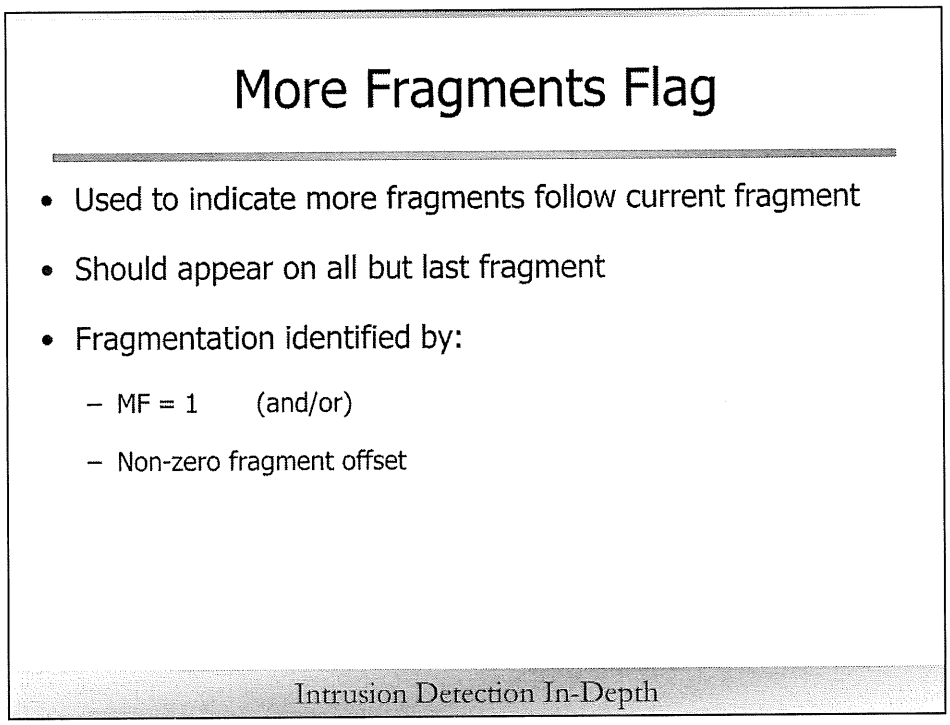


Let's examine the origin of the field that identifies fragments. The IP identification value is a 16-bit field found in the IP header of all packets. This uniquely identifies each packet sent by the host. This value may be incremented by 1 for each packet the host sends, although the trend now is to have TCP/IP stacks randomize this value. When the packet is fragmented, the host or router that fragments the packet will include the same IP identification number, or the fragment ID, in the IP header of each fragment to facilitate later reassembly.

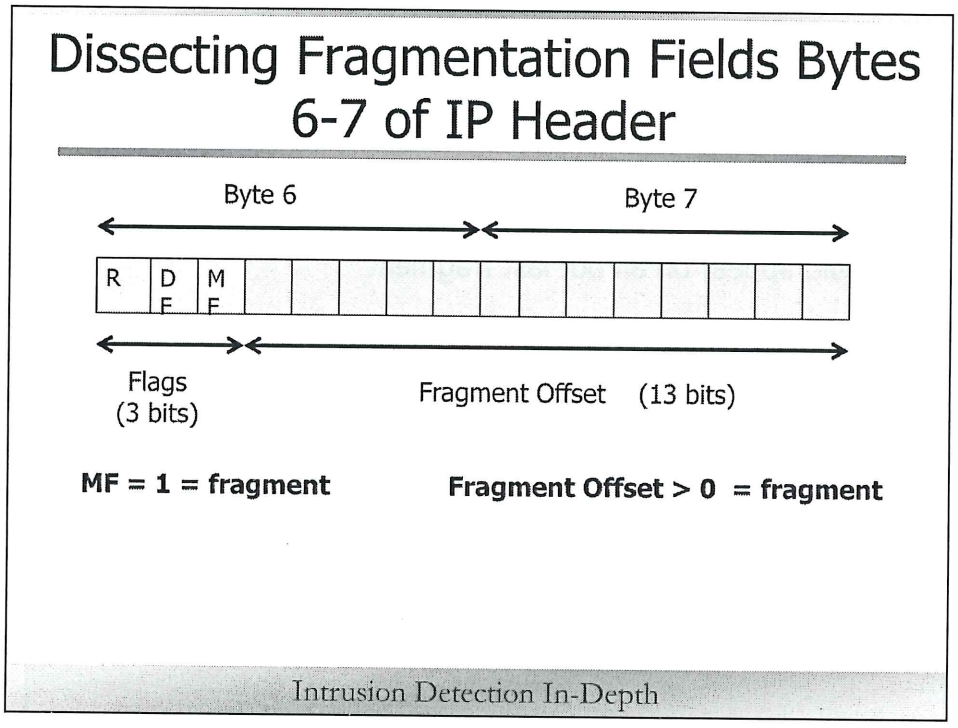


Earlier, we learned that the fragment offset field is a 13-bit field that expresses fragments as multiples of 8. Therefore every fragment has to be a multiple of 8.

There may be some confusion where in the packet to begin the offset count. If you examine the original unfragmented packet, the data that follows the IP header is considered the "fragmentable" part. So, the offset counting begins at the transport layer.



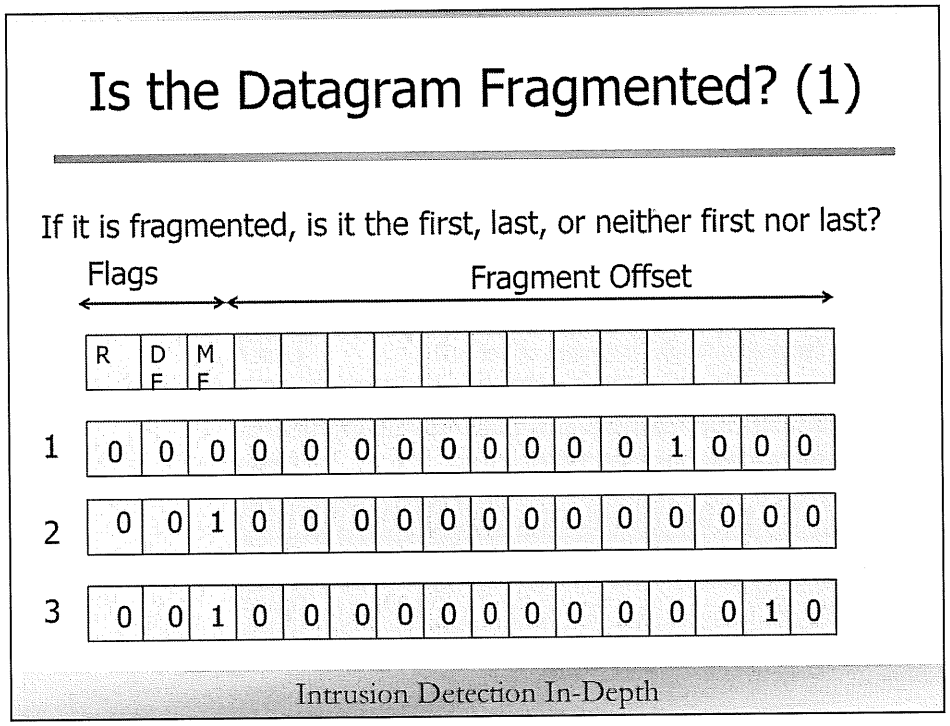
The more fragments (MF) indicates whether or not one or more fragments follow the current one. All fragments except the final one should have the MF flag set. The way that a receiving host will detect fragmentation is that MF flag is set and/or the fragment offset field in the IP header is non-zero.



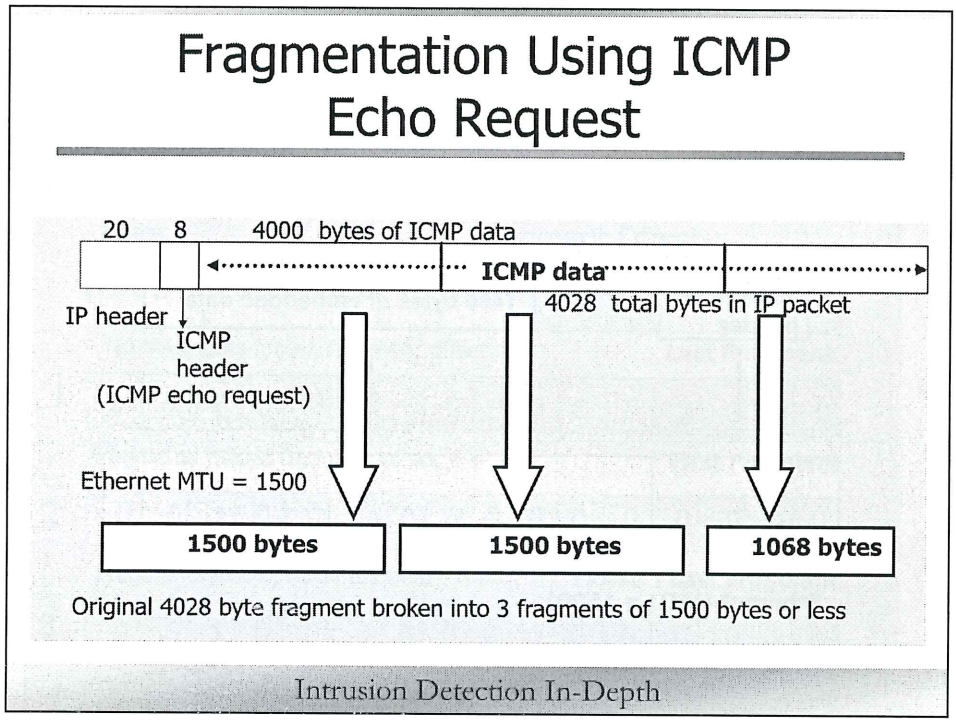
Let's revisit the IP header, but this time with an intent of detailed examination of the actual bytes and bits associated with fragmentation fields in the IP header to determine if a datagram is fragmented or not. The fields associated with fragments are located in the 6th and 7th bytes offset from the beginning of the IP header. The 6th byte is a combination of the high-order 3 bits for IP flags and the lower 5 bits for the fragment offset. The entire 7th byte, is assigned for fragment offset bits only.

As you can see the 13-bit fragment offset is split between bytes 6 and 7. We are really not concerned about the fragment offset value for this discussion, just the determination of whether or not the bits and bytes that are in these two bytes indicate that the datagram is fragmented. How would you go about ascertaining whether or not a particular datagram is part of a fragment train simply by looking at these two bytes? There are two different fields that need to be examined. The first is the MF bit found in byte 6. When set, it indicates that more fragments follow. This bit should be set in every fragment except the last. Next, the fragment offset found in the 13 bits split between bytes 6 and 7 indicates whether or not this is the first or subsequent fragments.

Here's a little theory before we put it into practice. If the MF flag is set or the fragment offset is non-zero, it is a fragment. If the MF flag is set and the fragment offset is 0, it is the first fragment. When the MF flag is set and the fragment offset is non-zero, you have a fragment that is neither the first nor the last – somewhere in between the two. And, finally when the MF flag is not set, yet the fragment offset is greater than zero, you have the last fragment.



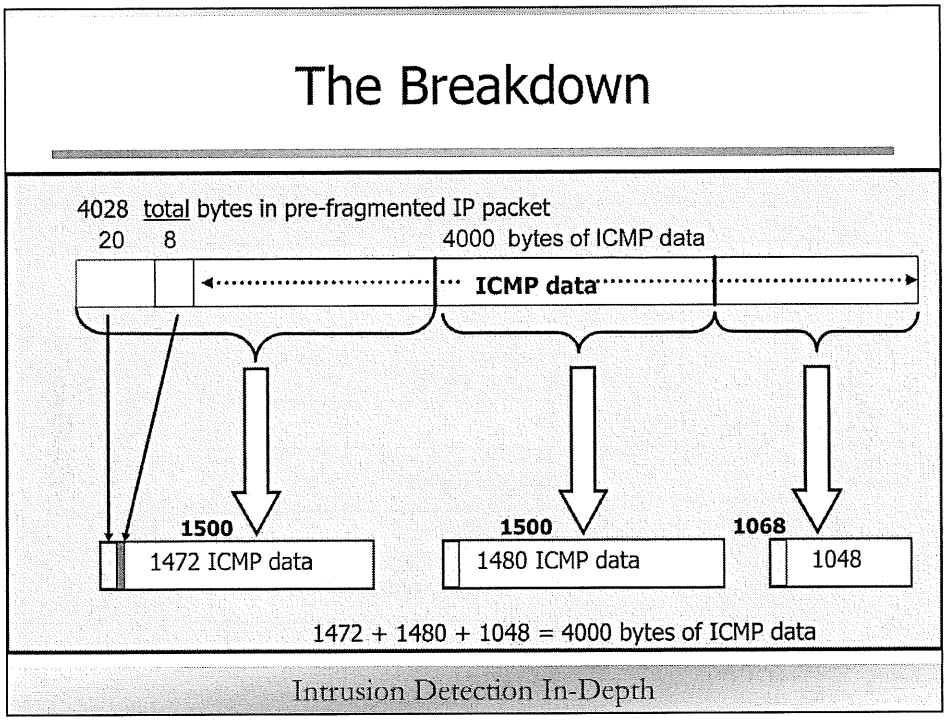
Let's test that theory with three example IP header bytes 6 and 7. Take some time to examine each of these examples to determine first whether or not the datagram is a fragment and second, the fragment's place in the entire fragment train - first, last, neither first nor last (somewhere between the first and last).



4000 + 8 + 20

Let's follow the process of fragmenting an oversized packet into fragments. Here we have a packet of 4028 bytes. This is an ICMP echo request bound for an Ethernet network that has an MTU of 1500. So, the 4028 byte packet will have to be divided into fragments of 1500 bytes or less. Each of these 1500 byte fragmented packets will have a 20 byte IP header, so that leaves 1480 bytes maximum for data for each fragment. We'll scrutinize in the next series of slides the values and data found in each of the created fragments.

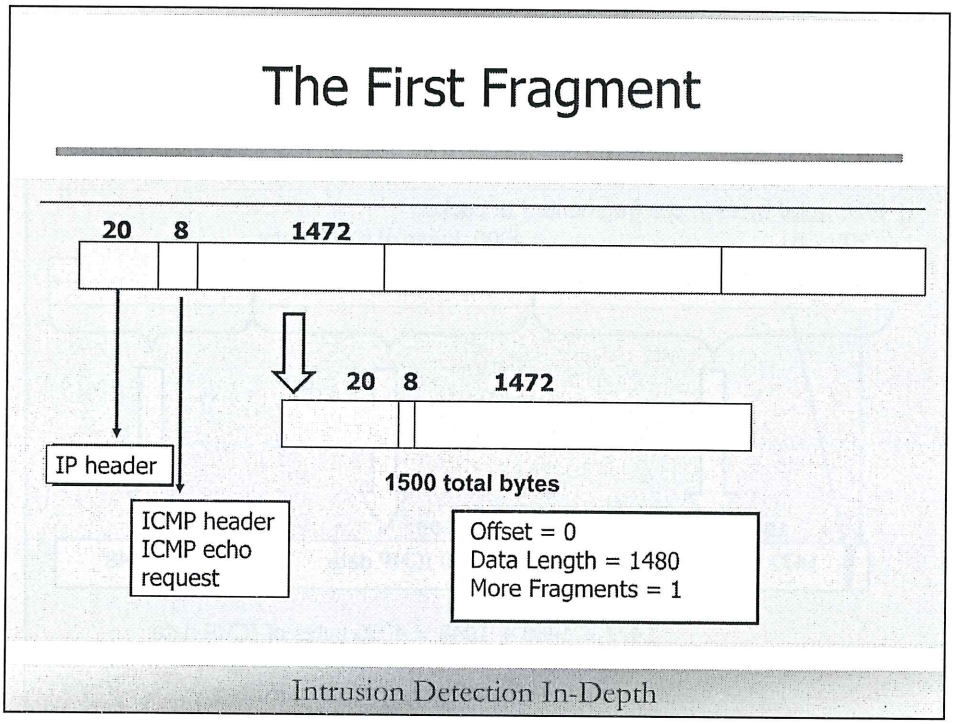
Normally, you shouldn't encounter a 4,000+ byte echo request. The reason that this was used for the example and for instructive purposes is to allow the generation and capture by tcpdump of the packets you see in the upcoming several slides.



Here is how each fragment is actually formed. Before the IP packet is sent on the link that has an MTU of 1500 bytes, we see that is has a total of 4028 bytes total.

What we have seen is that this IP packet is divided into three separate fragments each with a cloned IP header. Creation of the second and third fragments requires two new IP headers – each 20 bytes for a standard IP header. So, we really need a total of 4068 bytes to send all of this traffic.

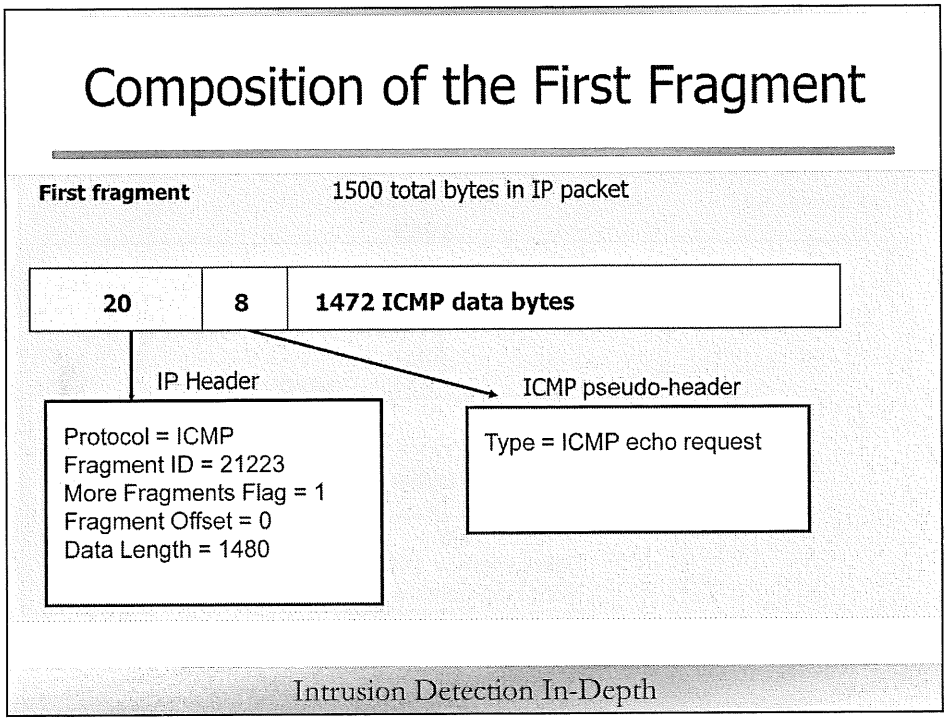
The first fragment gets a slightly modified original IP header, along with the 8 bytes of the ICMP header for a running total of 28 bytes. With a maximum packet size of 1500 bytes, 1472 bytes remain for ICMP data. The second fragment gets a cloned IP header of 20 bytes, and has the remaining 1480 bytes for ICMP data. The final fragment again gets another cloned 20-byte IP header and carries the final 1048 bytes of ICMP data. As a cross check, we see that we have 1500 + 1500 + 1068 bytes of data sent for a total of 4068 bytes.



Let's turn our concentration to the initial fragment in the fragment train. The "original" IP header will be cloned to contain the identical fragment identification numbers for the first and remaining fragments. Remember, all fragments must be carried in an IP packet. An IP packet requires an IP header to direct it to its destination.

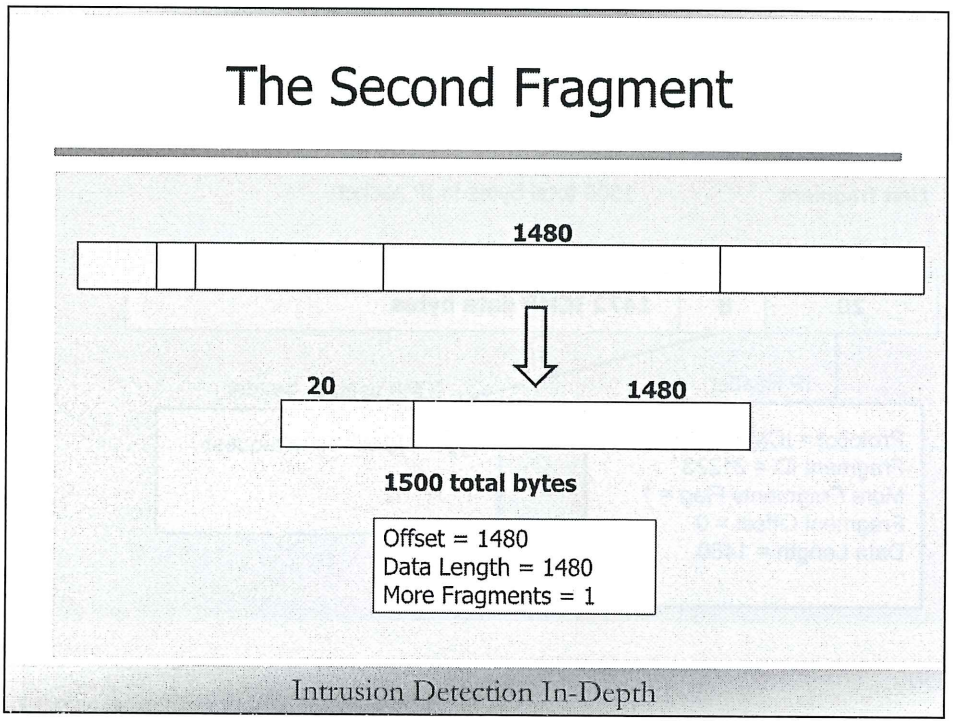
The first fragment is the only one that will carry with it the ICMP message header. This is true for all fragment trains created. The first fragment only carries the protocol header. All remaining fragments carry data only. This is important to remember when creating any kind of tcpdump or Wireshark filter to view fragments. If you erroneously attempt to filter all fragments in a given fragment train using a value found in the transport protocol header, you will see the first fragment only. For instance, suppose that a packet with TCP destination port 80 (HTTP) is fragmented. With normal fragmentation, the first fragment is the only one that contains the TCP header, including the destination port of 80. If you were to try to use a filter that selects fragments with a destination port of 80, you would see the first fragment only.

As we see, the first fragment has a 0 offset, a data length of 1480 bytes, 1472 bytes of data and 8 of ICMP header, and more fragments follow so that more fragments flag is set.



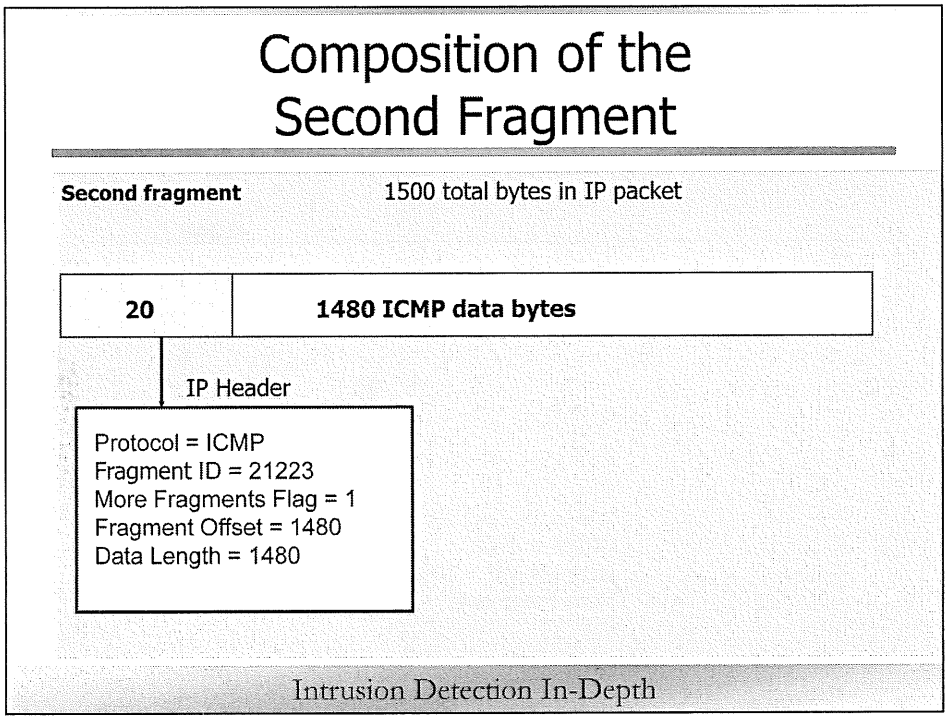
Examine the configuration of the first fragment in the fragment train. The first 20 bytes of the 1500 bytes are the IP header. The next 8 bytes are occupied by the ICMP header. Recall that this was an ICMP echo request that has an 8 byte header in its original packet. The remaining 1472 bytes are for ICMP data.

In addition to the normal fields carried in the IP header such as source and destination IP and protocol, in this instance, ICMP, there are fields that are specifically for fragmentation. The fragment ID with a value of 21223 will be the common link for all the fragments in the fragment train. The more fragments flag is set to 1 to indicate that more fragments do follow. Also, the offset must be stored of the data contained in this fragment relative to the embedded data in the prefragmented packet. For the first record, the offset will be 0. The 1480 bytes of data represent the 8 byte ICMP header followed by the first 1472 bytes of the ICMP data; this does not include the 20 bytes of IP header.



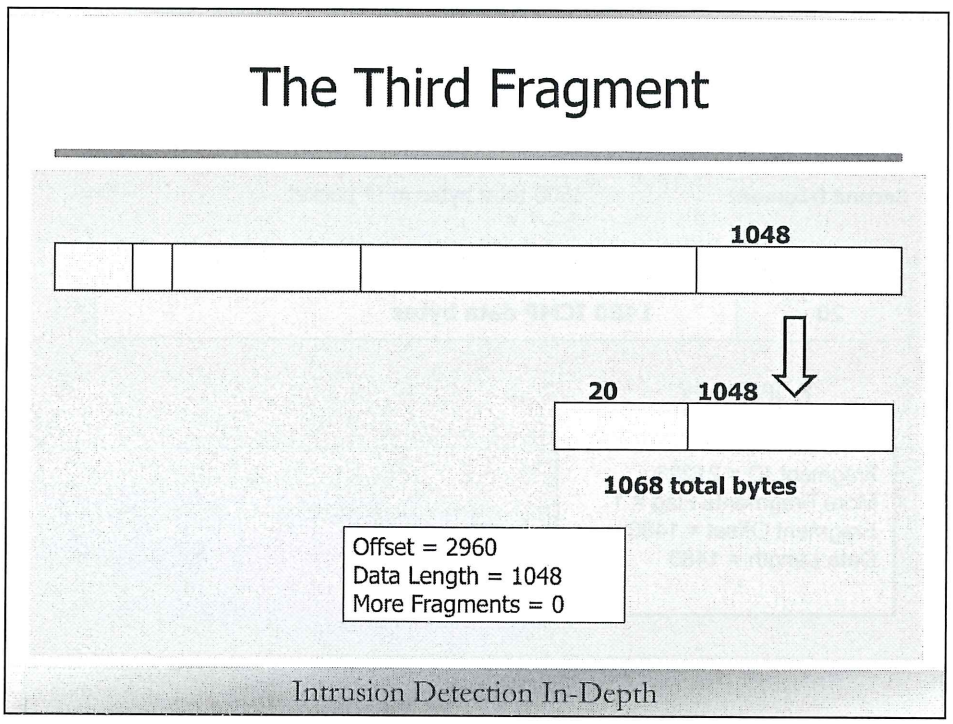
We focus on the next fragment in the fragment train. An IP header will be cloned from the original IP header with an identical fragment identification number, and most of the other data in the IP header such as the source and destination numbers will be replicated for the new header. Embedded after this new IP header will be 1480 ICMP data bytes. You may be wondering why the offset is 1480 instead of 1500 bytes. After all, the first fragment had a total of 1500 bytes. But, remember that the fragments are created from the data that follows the oversized packet's IP header. Therefore the size of the IP header of the first, and all subsequent packets is not considered part of this data and is not counted in the offset value.

As we see, the second fragment has an offset of 1480, a data length of 1480 bytes, and one or more fragment follow so the more fragments flag is set.



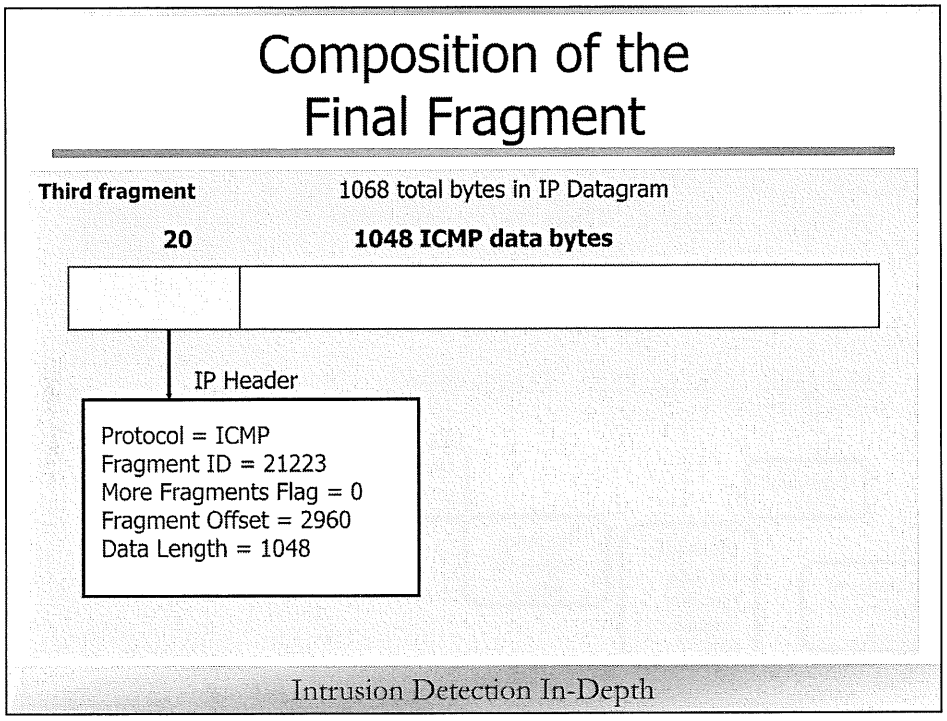
Continuing with fragmentation, we examine the IP packet carrying the second fragment. As with all fragments, it requires a 20-byte IP header. Again, the protocol in the header will indicate ICMP. The fragment identification number remains 21223. And, the more fragments flag is turned on because at least one more fragment follows. The offset is 1480 bytes into the data portion of the original ICMP message data. The previous fragment occupied the first 1480 bytes of ICMP header and data. This fragment will have 1480 bytes of data as well and it is composed entirely of ICMP data bytes.

Remember that the ICMP header in the first fragment doesn't get copied along with the ICMP data is subsequent fragments. This means if you were to examine this fragment alone, you could not tell what the ICMP message type is – in this case, an ICMP echo request. However, since each fragment inherits a cloned IP header, you can tell the embedded protocol of each fragment – in this case ICMP.



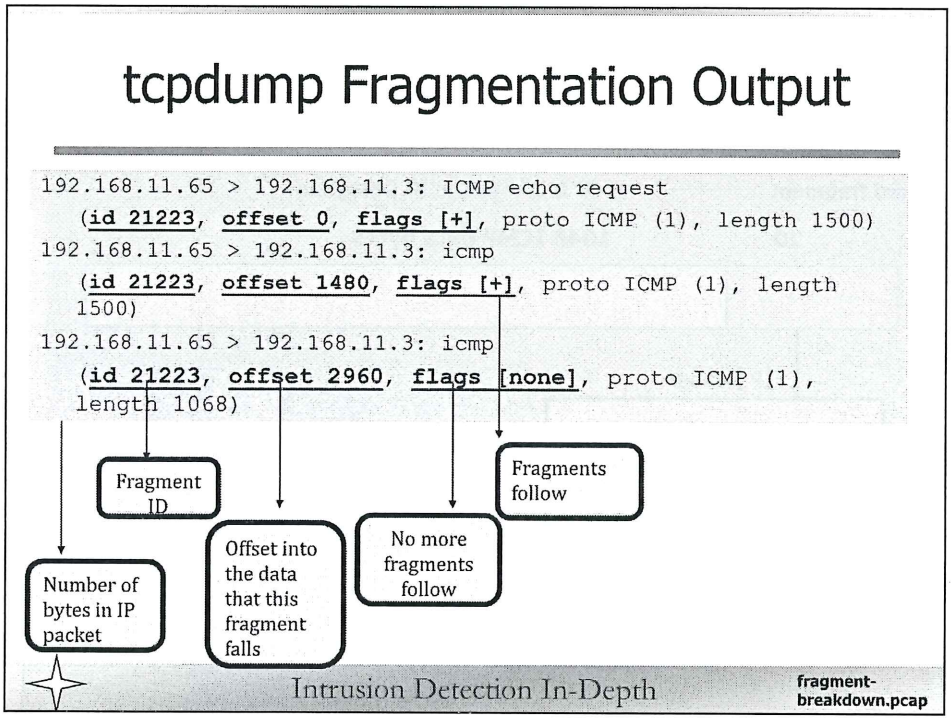
Finally, examine the third and last fragment in the fragment train. Again, an IP header is cloned from the original header with an identical fragment identification number, and other fields will be replicated for the new header. Embedded in this new IP packet will be the final 1048 ICMP data bytes.

As we see, the third fragment has an offset of 2960, a length of 1048 bytes, and no more fragments follow, so the MF flag is 0.



Here is a depiction of the last fragment in the fragment train. Again, 20 bytes are reserved for the IP header. The remaining ICMP data bytes are carried in the data portion of this fragment. The fragment ID is 21223, the more fragments flag is not set because this is the last fragment. The offset is 2960 (this is the sum of the two 1480 byte previous fragments). There are only 1048 data bytes carried in this fragment comprised entirely of the remaining ICMP message bytes.

This fragment, like the second one, will have no ICMP header and therefore no ICMP message type to reflect that this is an ICMP echo request.



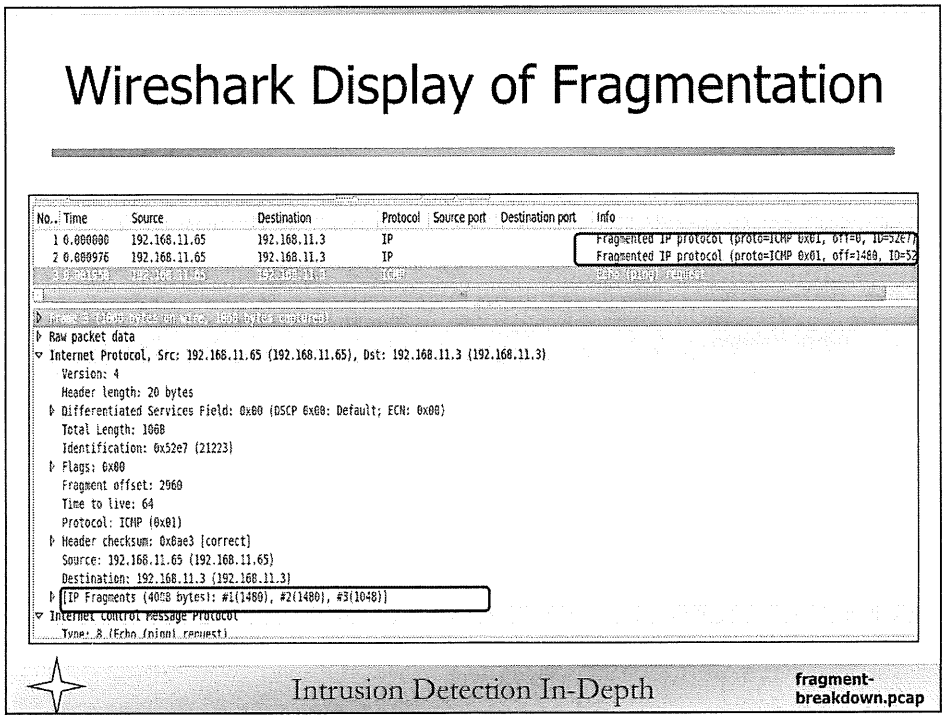
The above output was created using the tcpdump verbose option (-v), however the format was edited to make it more readable and coherent.

The first line shows 192.168.11.65 sending an ICMP echo request to 192.168.11.3. The reason that tcpdump can identify this as an ICMP echo request is because the first fragment contains the 8-byte ICMP header which identifies this as an ICMP echo request (type 8, code 0). Now, let's look at fragmentation notation from tcpdump. The verbose option displays the fragment/IP ID value of 21223. The fragment offset is apparent, and "+" is used to signify that the "more fragments" bit is set or "none" to indicate it is not. The length of the entire packet including the IP header is shown - in this case it is 1500 bytes – 20 bytes for the IP header and 1480 bytes for fragment data.

The second record is somewhat different. Notice that there is no ICMP echo request label. This is because there is no ICMP header to tell what kind of ICMP traffic this is. This fragment contains only ICMP data. The IP header still has the protocol field set to ICMP if you were to examine the 9th byte offset of the IP header, you'd find a value of 0x01 for ICMP. Yet that is all you can tell about the embedded protocol looking at this fragment alone. The fragment ID is 21223, and the offset is 1480. This is a little confusing since the first fragment had a length of 1500, but remember it includes a 20-byte IP header. The displayed value of 1480 represents only the number of bytes that follow the IP header. More fragments follow and the length of this fragment again including the IP header is 1500 bytes.

The last line is very similar to the second one in format. It shows the same fragment ID of 21223, and an offset of 2960 - the cumulative total of bytes from the two 1480 fragments before it. As you can see, there are no more fragments that follow since "flags[none]" means this is the final fragment. It has a length of 1068 – 20 bytes of IP header and 1048 of fragment data.

To see the output similar to that above, enter at the command:
tcpdump -nvt –r fragment-breakdown.pcap



This is the way that Wireshark displays that same set of fragments. Wireshark does some interpretation of the fragments. You may find this helpful or confusing depending on your point of view. The tcpdump interpretation is pretty straightforward, giving you the necessary detail and allowing you to perform the "translation" of the fragments. Wireshark, however takes the liberty to perform some of the translation, omitting some details and requiring you to rely on it for the interpretation.

You see the same three fragments from tcpdump. The ID number is expressed as the hex value 0x52e7 instead of the 21223. This is strictly a matter of preference whether you want to see the value in decimal or hex. Either way the IP ID is consistent. But, let's take a look at some of the other fields. For instance, if you look at the Protocol field column you see that the first two packets have a protocol of "IP", while only the last one has a protocol of "ICMP". This really is not correct – all three fragments carry the value of 1 in the protocol field, designating ICMP. The Info column correctly details the protocol.

Also, Wireshark offers the reassembly of all fragments in line 3. It interprets the reassembled fragments as an ICMP echo request. Recall that the protocol header is found in the first fragment only so it would be more accurate to have this interpretation on line 1. While Wireshark intends to be helpful and deliver you what it considers the pertinent data and translation, it takes license in its interpretation. Perhaps you don't care about the details and this is sufficient for your purposes. However, the interpretation that tcpdump offers is more esoteric to the novice, yet it is also more pure in that it doesn't get overly zealous in translation.

To see the output above, enter the following at the command line:

wireshark fragment-breakdown.pcap

Miscellaneous Questions About Fragments (1)

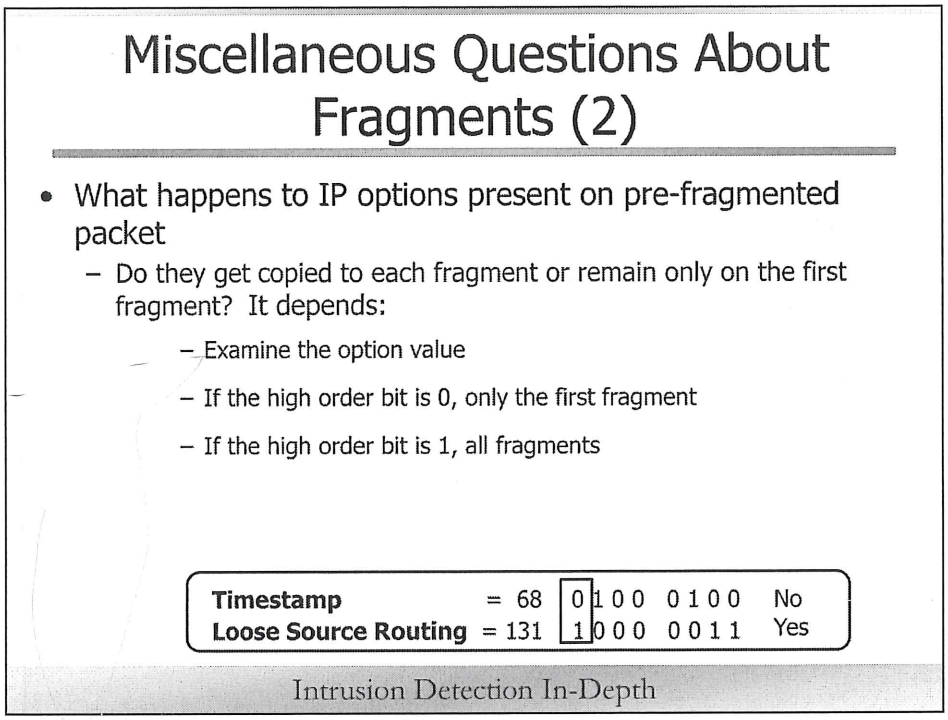
- What about the fragment(s) length?
  - The original IP header had the length of the pre-fragmented packet
  - Each fragment gets its own new length of the IP header bytes + what follows
- What about the fragment(s) checksums?
  - Each fragment gets its own new IP checksum
  - First fragment only has the embedded protocol checksum
  - The embedded protocol checksum is performed on the pre-fragmented protocol and data

Intrusion Detection In-Depth

What happens to the IP header datagram length value with fragments? As you are aware the pre-fragmented IP header contains a length for the number of bytes in the IP header plus all that follow. This is replaced in each fragment with the IP header length of the fragment plus the bytes that follow in that fragment only.

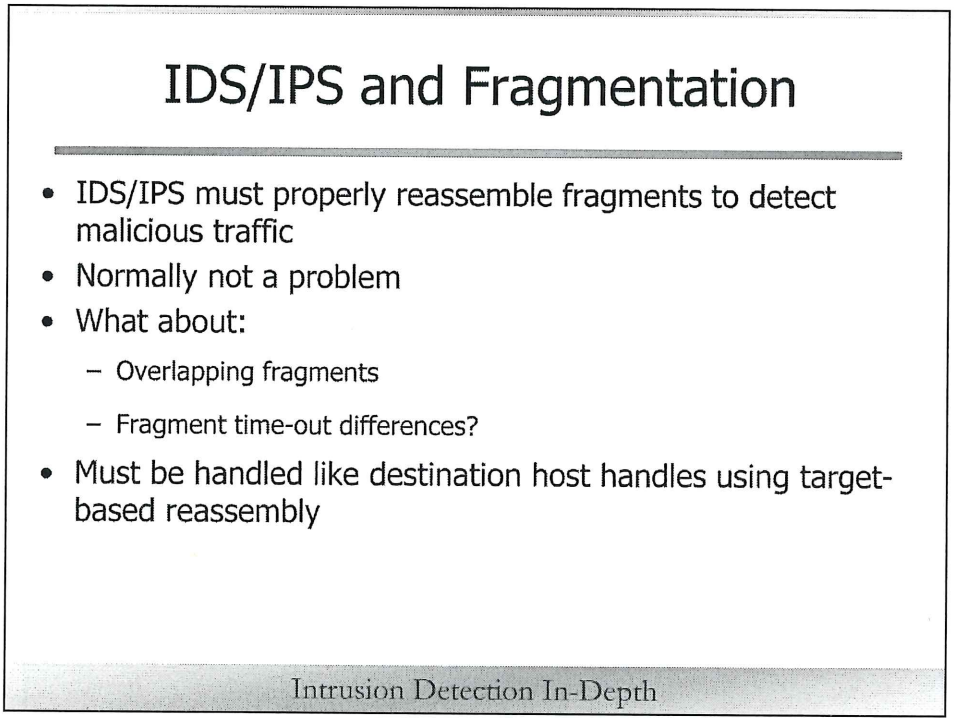
The concept of the IP checksum does not change. Since each fragment essentially gets a new unique IP header designating its offset value in the fragment train and whether or not any fragments follow, each IP header has a unique IP checksum computed for its fragment header.

While we have not discussed embedded protocol checksums such as TCP, UDP, and ICMP, they each have their own checksums stored in the respective TCP, UDP, or ICMP header. Since the protocol header appears in the first fragment only of normal fragmentation, the embedded checksum protocol value remains unchanged.



Suppose there are one or more IP options and a packet that needs to be fragmented. Do those IP options accompany each fragment or just the first? It depends. It seems that whoever created the numbering scheme for the value of any given IP option had forethought about this situation. Any IP option value that has a 1 in the most significant bit of the 1-byte field has all the IP options present in every fragment. In other words, any IP option value that is greater than 127 accompanies every fragment. Conversely, an IP option value that has a 0 in this same bit accompanies the first fragment only.

Take the Timestamp option with a value of 68. This option captures the timestamp for each hop of its journey. Apparently, maintaining the timestamp values for the first fragment only appears to be sufficient. However, the loose source routing option value of 131 requires all the fragments to keep the routing information. If you remember, the loose source routing option suggests a series of routers through which the packet traverses. It makes sense that all fragments must maintain that data.

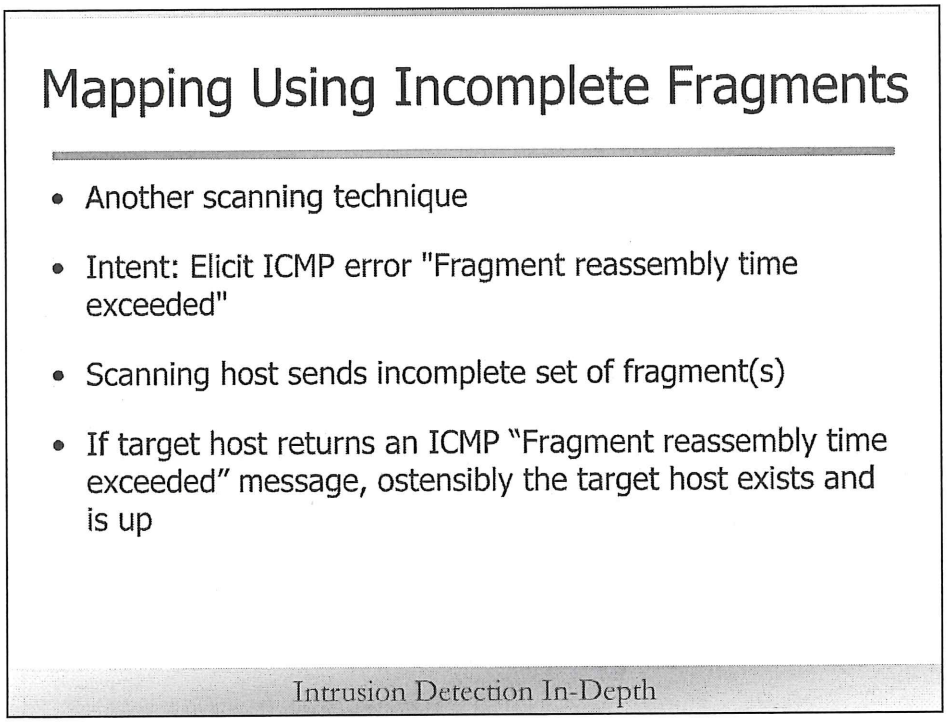


Since reassembly of fragments is absolutely vital for an IDS/IPS to examine the entire packet, it is imperative for the IDS/IPS to do it properly. But, what is properly? Most of the times proper reassembly means that the IDS/IPS must reassemble all fragments that have the same source and destination IP addresses, protocols, and identical IP identification numbers using the fragment offset values found in the IP header of the fragments. This is a trivial task for the IDS/IPS to perform.

But, an attacker can craft traffic to deliberately introduce ambiguities for an IDS/IPS. For instance, what about fragments that overlap? How should it handle two fragments that have identical offsets, but contain different payload. Does the IDS/IPS honor the first or second fragment? What if it chooses to accept and evaluate the first fragment that has innocuous payload and ignores or does not block the second fragment that contains an exploit payload? The damage depends on whether the destination host accepts the first or second fragment. If it accepts the first fragment, no harm is done. However, if it accepts the second fragment and the exploit is successful, the IDS or IPS has failed to do its job.

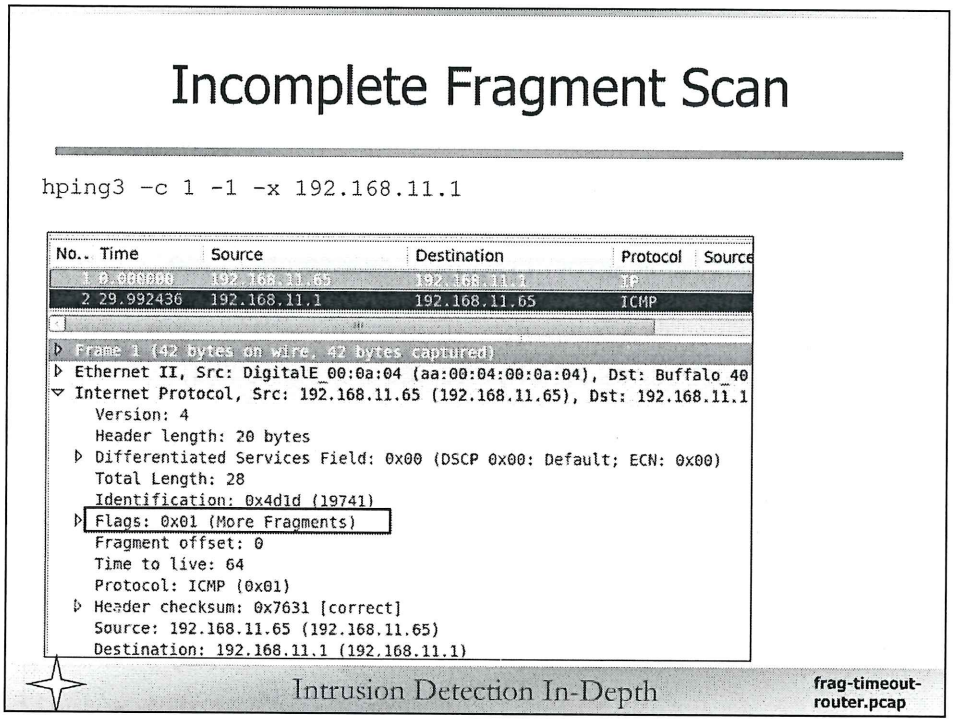
Consider a different situation that involves a crafted set of fragments where one is late or delayed. Some operating systems may wait for over a minute for a delayed fragment to arrive before timing them out. What if an IDS or IPS has a shorter wait time and flushes the fragments before the destination host? This also poses a potential threat to that IDS/IPS. A busy IDS/IPS that handles possibly gigabytes of traffic may not have the luxury of allowing a minute's worth of idle activity for fragments since it needs to be stingy with memory resources. If the IDS/IPS does not handle fragments as the destination host does, it is possible for the delayed fragment to contain an exploit payload and compromise the destination host.

The solution to this problem and other TCP/IP related ambiguities is for the IDS/IPS to be target-based. In other words, it must know what the destination host is and how it reacts to overlapping or delayed fragments. The IDS/IPS must behave identically to the destination host in reassembling fragments, otherwise it can be evaded.



A mapping technique to find active hosts on a remote network is to try to elicit an ICMP "IP/fragment reassembly time exceeded" message from hosts on a scanned network. This method may be used instead of more conventional scans, such as an ICMP echo request, because those methods may be blocked from network entry. This can be done by sending an incomplete set of fragments to hosts that are being mapped. For this to work properly, the destination host has to be listening on the port or protocol that is used. If it does, when it receives the first fragment (not necessarily the zero offset fragment), it will set a timer.

If the timer expires and the receiving host has not received all the fragments, it will send the ICMP "Fragment reassembly time exceeded" error back to the sending host. It is important to note (according to RFC 792) that in order for the ICMP "Fragment reassembly time exceeded" error to be generated, the zero offset fragment must not be the missing one. RFC1122 recommends that the timer expire between 60 seconds and 2 minutes, though we'll see in the next slides that is not always the case.

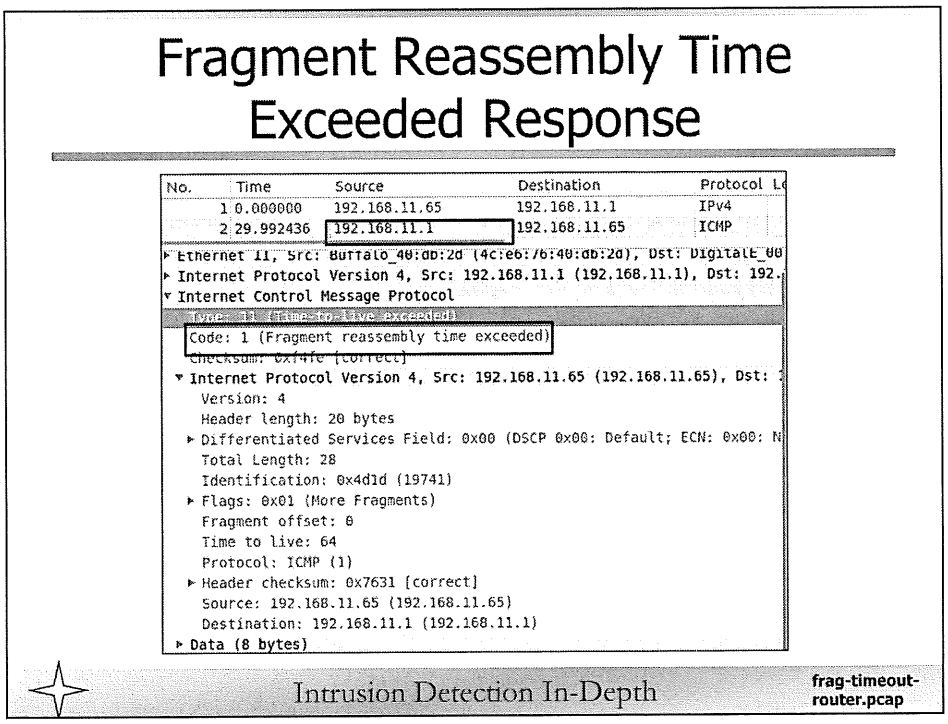


To simulate this scan, the open source packet crafting tool hping3 command is enlisted. The –c 1 option tells hping3 to generate one packet, otherwise it will continue until you stop it. The -1 options designates the use of ICMP - by default, an echo request, and the –x option to set the more fragment flag. Our target host is 192.168.11.1. Yet, we send no more fragments.

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

wireshark frag-timeout-router.pcap

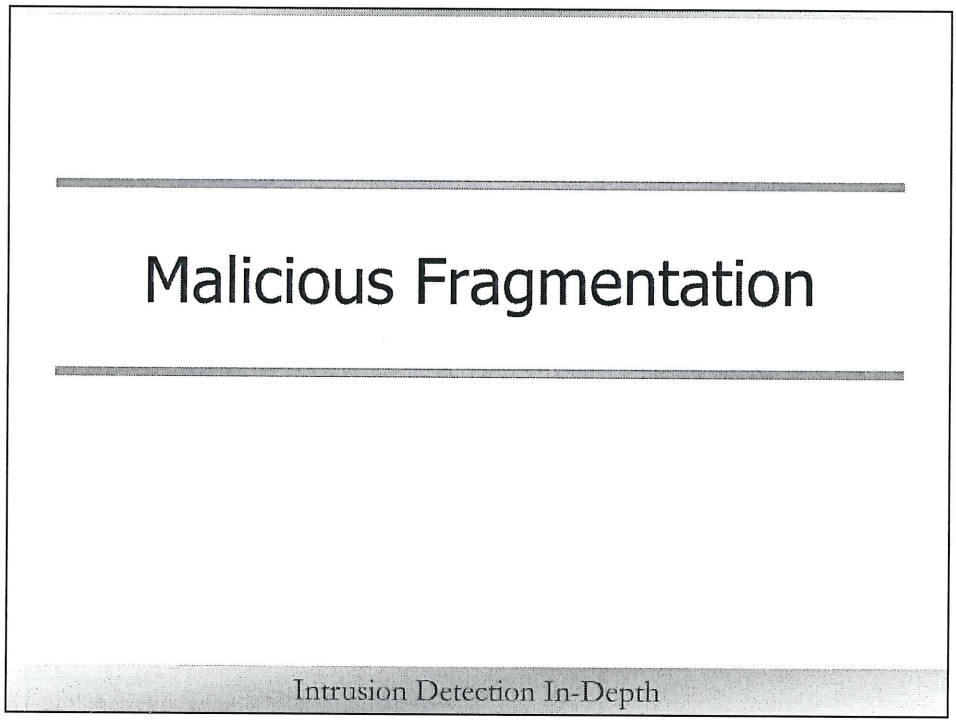
The first of the two records is shown above.



After almost a half a minute (look at the time field in the packet pane) elapses before 192.168.11.1 sends back an ICMP error message of "Fragment reassembly time exceeded". We'll cover ICMP formats and messages later in the course. However, you may notice that following the ICMP error message, there appears to be another packet embedded in the ICMP message. As we'll learn, the host that sends the error includes part of the packet that caused the error to occur. This gives the message some context, otherwise the recipient host of the error message would have a hard time pairing it with a stimulus packet. This takes all the guesswork out of it.

It is also possible that sending an incomplete set of fragments can help to identify the target host remotely. A 30-second timeout is common and would not identify a unique OS, but perhaps paired with other reconnaissance or probes, might serve to validate a guess or supplement some other data.

This is the second record of the file **frag-timeout-router.pcap**.



We now examine the topic of fragmentation used for purposes other than the intended ones.

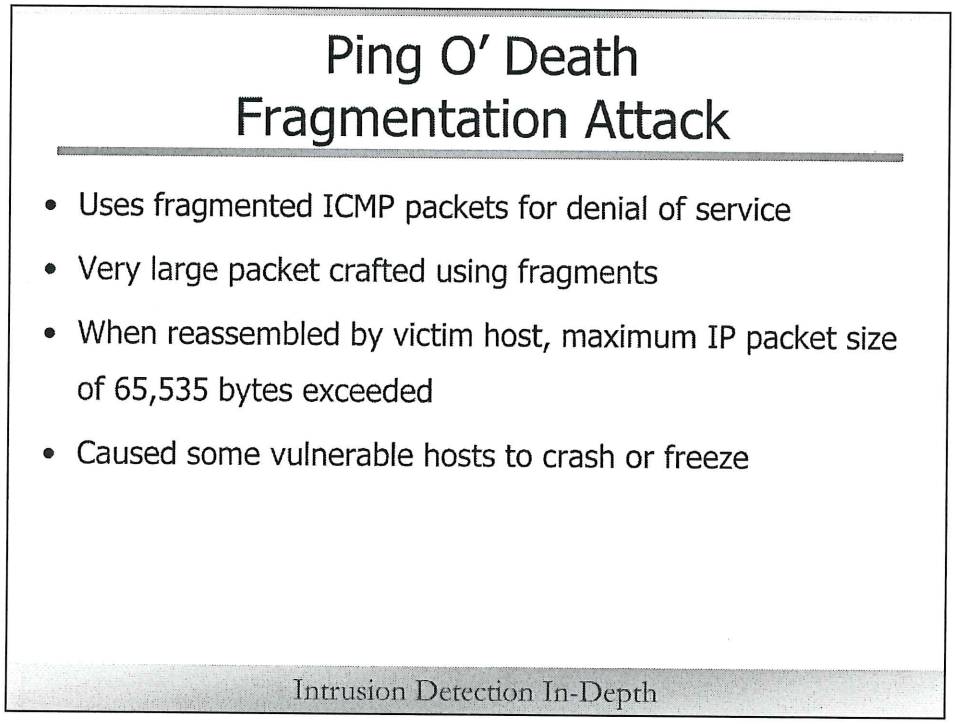
Fragmentation Attacks

- Many fragmentation attacks first appeared in mid to late 1990s, but still resurface today
- This is about the time that the Internet became available to the masses
- Attackers have discovered most IPv4 fragmentation attacks
- Current operating systems hardened against them

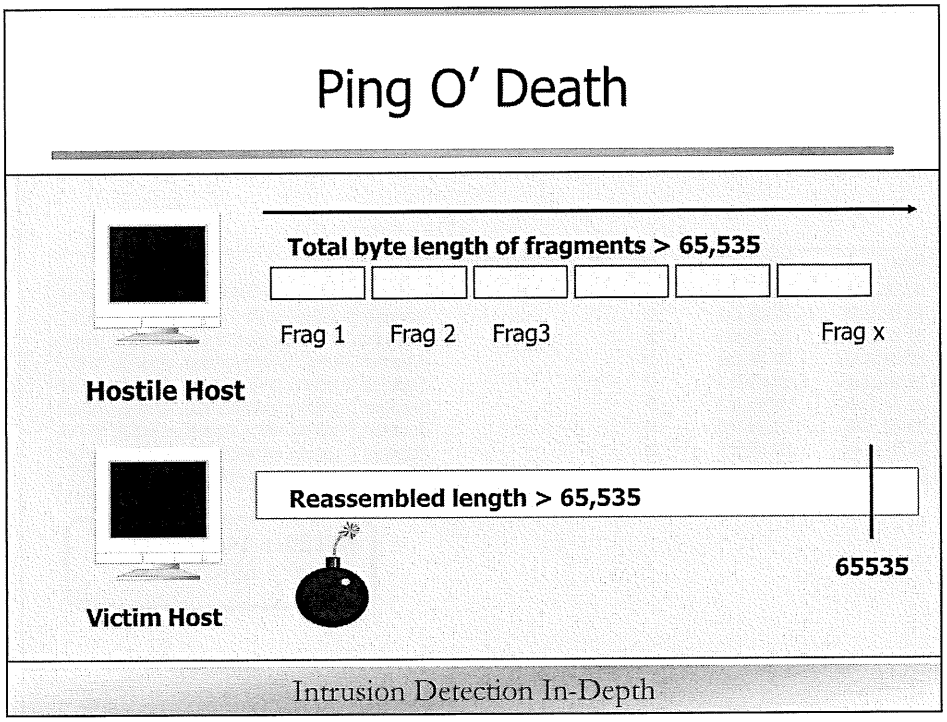
The IPv4 fragmentation attacks that are discussed in the upcoming slides occurred in the mid to late 1990s. Coincidentally, this is about the same time that the Internet became available to just about anyone with a computer, a modem, and dial-up access. Hackers quickly developed all kinds of attacks. Some favorite ones were denial of service attacks using fragmentation in unexpected and harmful ways.

At the same time, developers and maintainers of the most popular operating systems of the day had to combat all of the new, curious users of the Internet and their devious attacks. They more thoroughly examined their code and supplied patches to prevent denial of service attacks using fragments. It is important to be familiar with the history of and evolution of these fragmentation attacks to understand how attackers took advantage of the flaws.

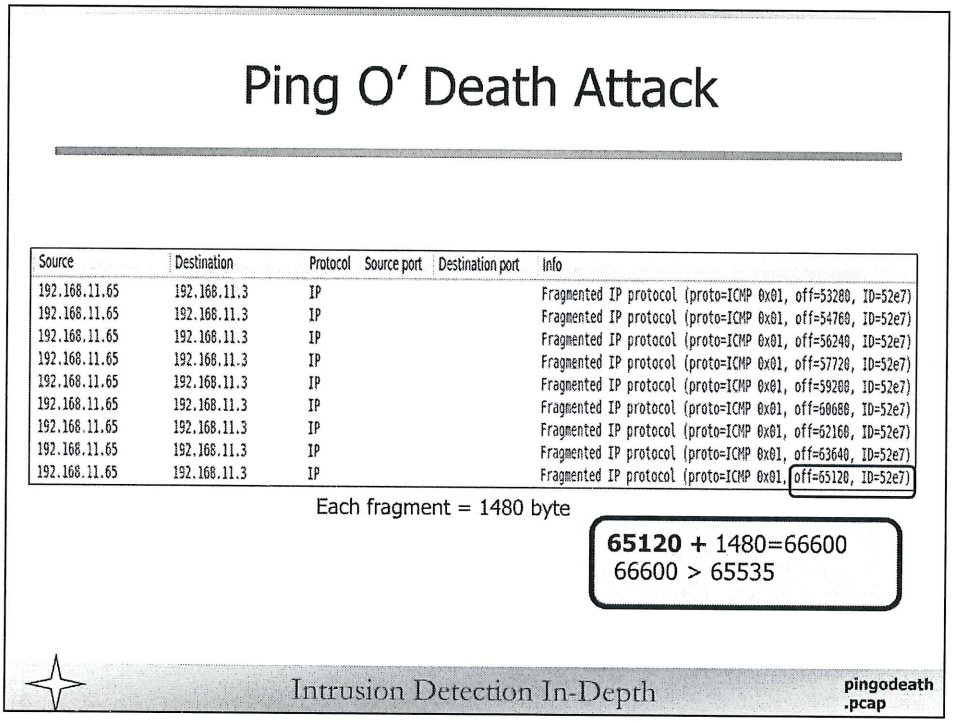
Attackers may still try these attacks. We're seeing older attacks resurfacing in IPv6 testing to see whether or not newer implementations of protocols are hardened against these attacks.



The Ping O' Death fragmentation attack is a denial of service attack that used a ping command to create an IP packet that exceeded the maximum 65535 bytes of data allowed by the IP specification. The oversized original packet was fragmented and then sent to a victim host. Some older operating systems would crash, hang, or reboot when they received such a maliciously crafted packet. This attack is not new, and all OS vendors should have fixes in place to handle the fragment reassembly of oversized packets.



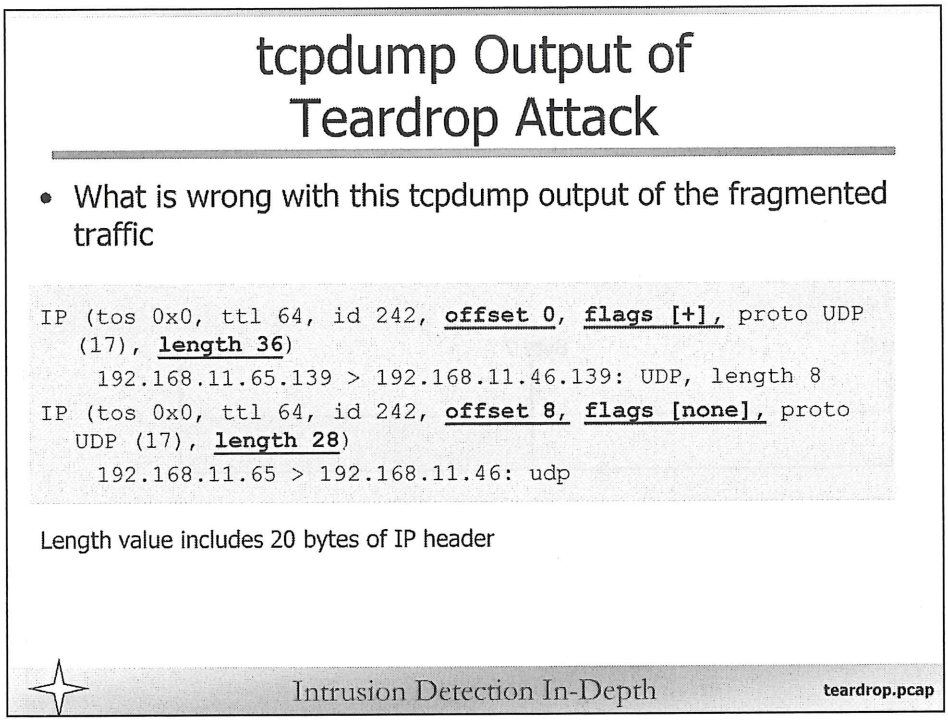
In the pictorial representation of Ping O' Death, we see a hostile host crafting an oversized IP packet from smaller fragments. When the victim host received these fragments and attempted to reassemble them, it may have experienced a denial of service failure when its reassembled packet length exceeds 65535 bytes.



This is how Wireshark interprets Ping O' Death traffic. We have a view of the packets right before the value of the reassembled packet exceeds the maximum size of 65535. Each of these fragments is 1480 bytes in length. Thus, when the packet depicted at the bottom is sent with an offset of 65120 and an additional 1480 bytes, a receiving host that was vulnerable to the Ping O' Death would crash or freeze.

To view the output, enter in the command line:

wireshark pingodeath.pcap

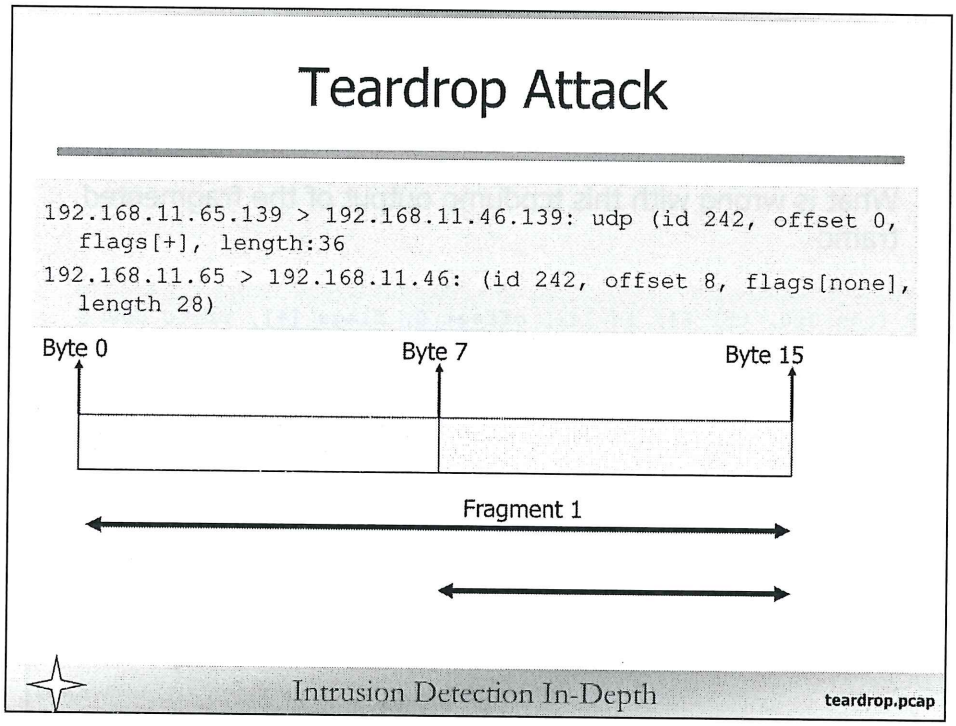


Now we see another earlier type of denial of service using UDP. The Teardrop attack exploited weaknesses in the reassembly process of fragments. The Teardrop program created fragments with overlapping offset fields. When these fragments were reassembled at the destination host, some older operating systems will crash, hang, or reboot. Again, this attack has been around for several years so patches should be available for vulnerable systems.

The first fragment has a length of 36. This includes a 20-byte header, 8-byte UDP header, and 8 bytes of payload. This means that the final 16 bytes are the fragment payload. The next in-order fragment should have an offset of 16. However, look at the offset of the second fragment – it is 8. This means that the two fragments overlap for 8 bytes of data because the second fragment has 20 bytes of IP header and 8 bytes of data that begin at offset 8 bytes into the first fragment.

To view the output, enter the following in the command line:

tcpdump –r teardrop.pcap -nvt

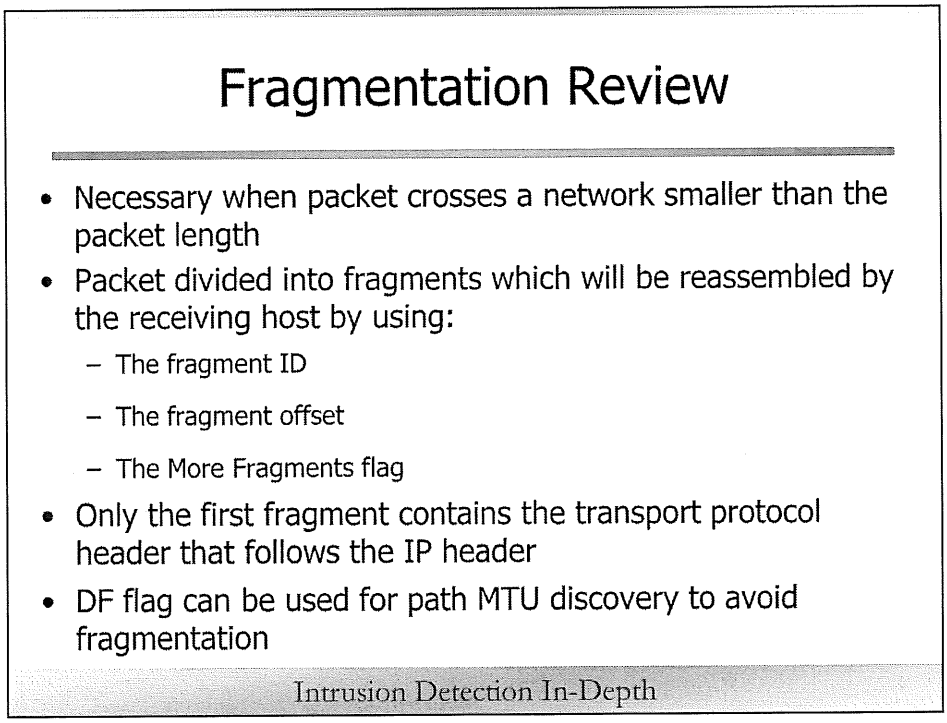


Let's examine the tcpdump output from Teardrop fragmentation alongside a depiction of it. The first fragment delivered is UDP and has a fragment ID of 242, a length of 36 data bytes and an offset of zero. It spans bytes 0 through 15 inclusive.

Now, the second fragment comes along. It is associated with the first fragment because of fragment ID of 242, it has a length of 28 – 20 bytes of header and 8 bytes of fragment payload. It begins at an offset of 8 bytes into the payload portion. As you can see, it actually overlaps offset bytes 7 through 15 of the first fragment.

To view the output, enter the following in the command line:

tcpdump –r teardrop.pcap -nvt

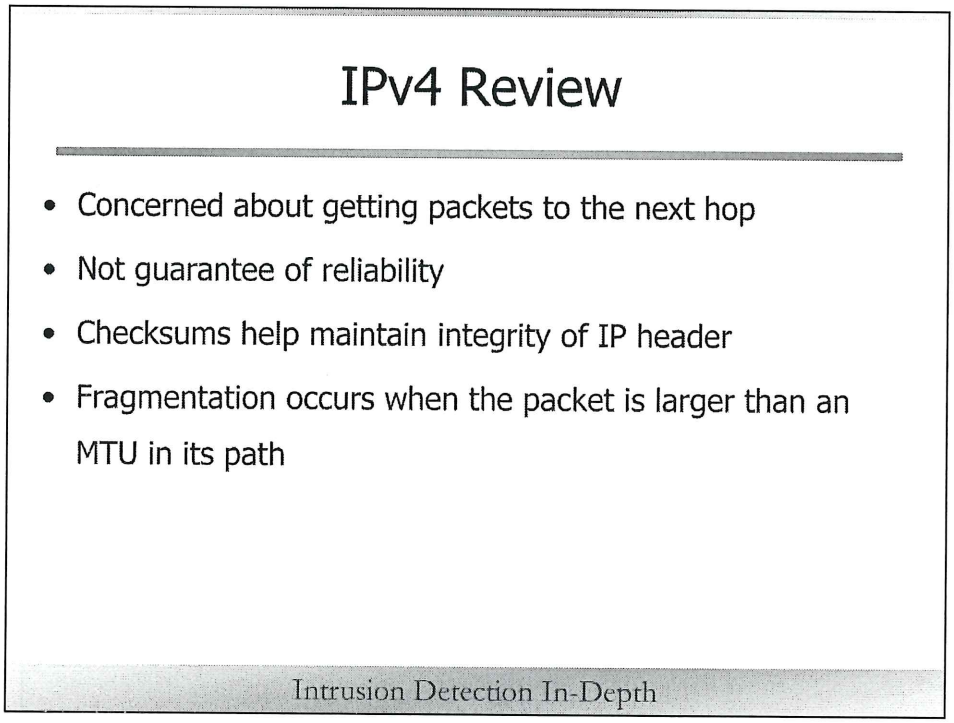


We've seen where fragmentation is a normal occurrence for a packet travelling from a larger to a smaller network. If a packet requires fragmentation along the path to the destination, a router on the network with the smaller MTU will fragment the packet if the Don't Fragment flag is not set.

Each fragment is encapsulated in an IP packet. Every IP header contains information such as the fragment ID, the offset, and whether other fragments follow. Remember that only the destination host reassembles the fragments. No intermediate routing devices should reassemble them. However, an IDS/IPS must reassemble them to make sure that malicious content is not spread between fragments. The DF flag can be used as a mechanism to discover the MTU to the destination host and provide more appropriate packet packaging thus avoiding fragmentation all together.

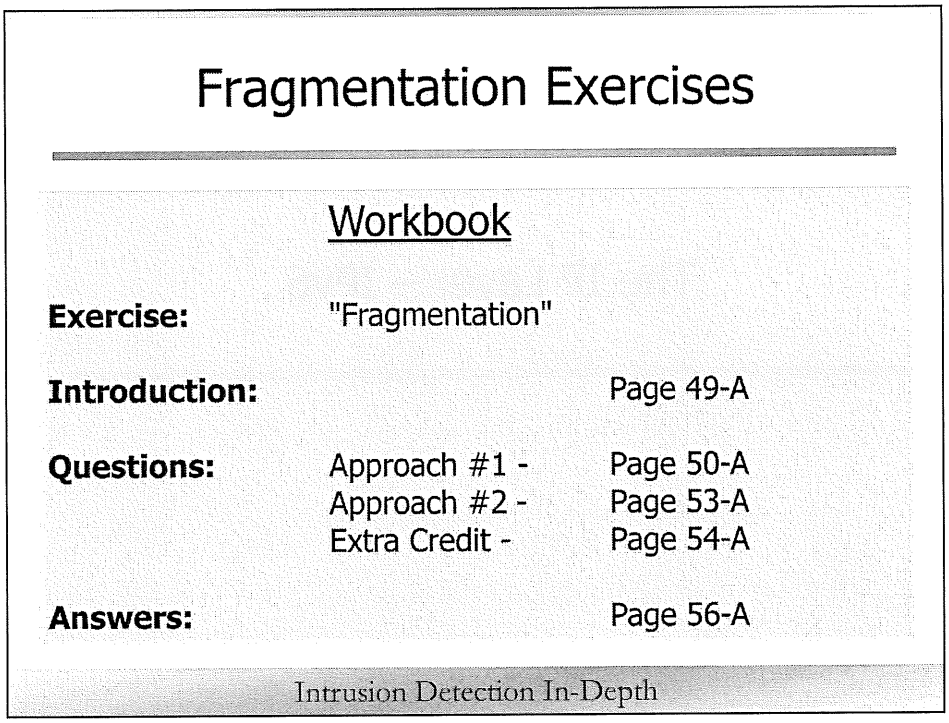
Remember that only the first fragment contains the transport layer header. All other fragments contain data only. This is important to remember when you are filtering fragments. If you specify a filter that includes a field and associated value from the protocol (ICMP, TCP, UDP), you will probably be baffled why you received the first fragment only.

One final point to remember is that IP is not a reliable protocol and it is possible for one or more fragments to get lost. In fact, if one or more fragments does not arrive at the destination, all must be re-sent. The receiver begins a fragment timer – different operating systems have different values ranging from 30 seconds to over a minute – when a fragment arrives. The receiver has to inform the sender if all fragments do not arrive when the timer expires to make the sender aware that it needs to resend all the fragments. The receiver informs the sender using ICMP – specifically an "IP/fragment reassembly time exceeded" message.

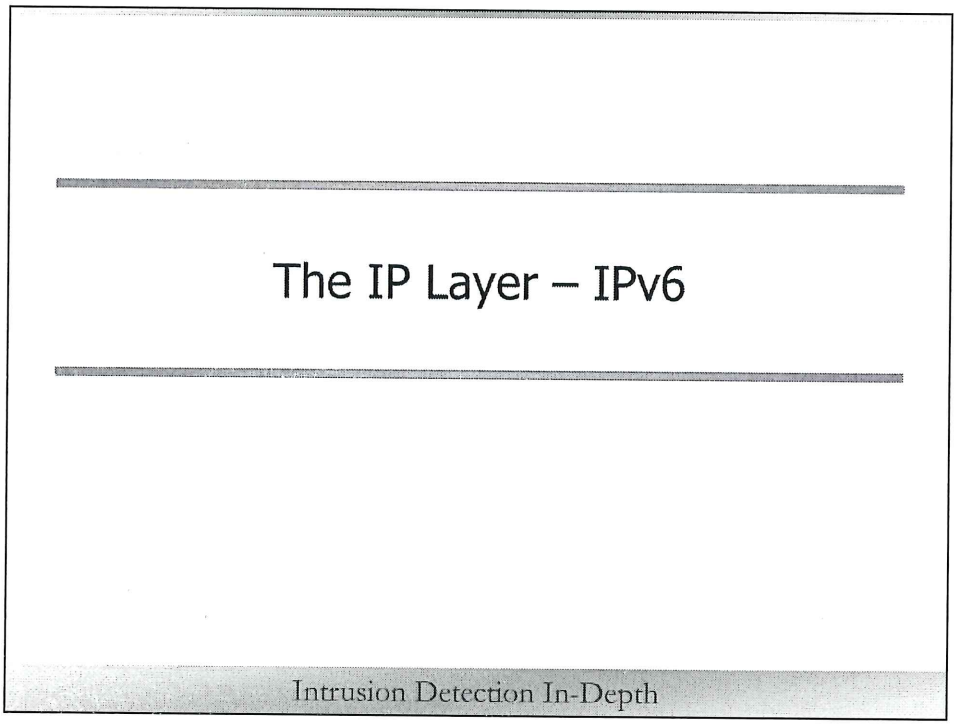


The main function of the IP layer is to move packets from one hop to the next. IP has no means to guarantee reliability. Instead, upper layer protocols – transport or application layers – must include some means to assure that packets make it to their destination. Checksums validate that the data in the IPv4 header does not get corrupted in transit, using a simple formula whose result is placed in the IPv4 header and must be validated by routers en route and the receiver to assure the integrity of the data.

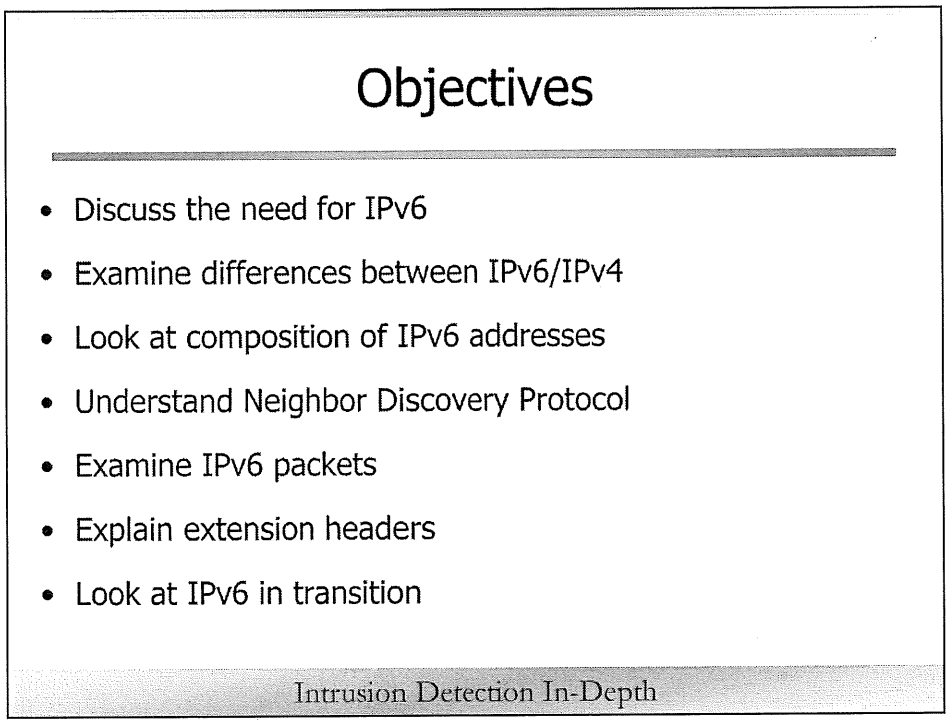
Fragmentation occurs when a packet is larger than the next link layer MTU. Fragmentation causes some overhead because it creates several different packets and if any fails to reach the destination, all must be sent again. It is best for a host to avoid this in the first place. That may be done using path MTU discovery to find the smallest MTU from source to destination and format all packets to fit within that MTU length.



This page intentionally left blank.



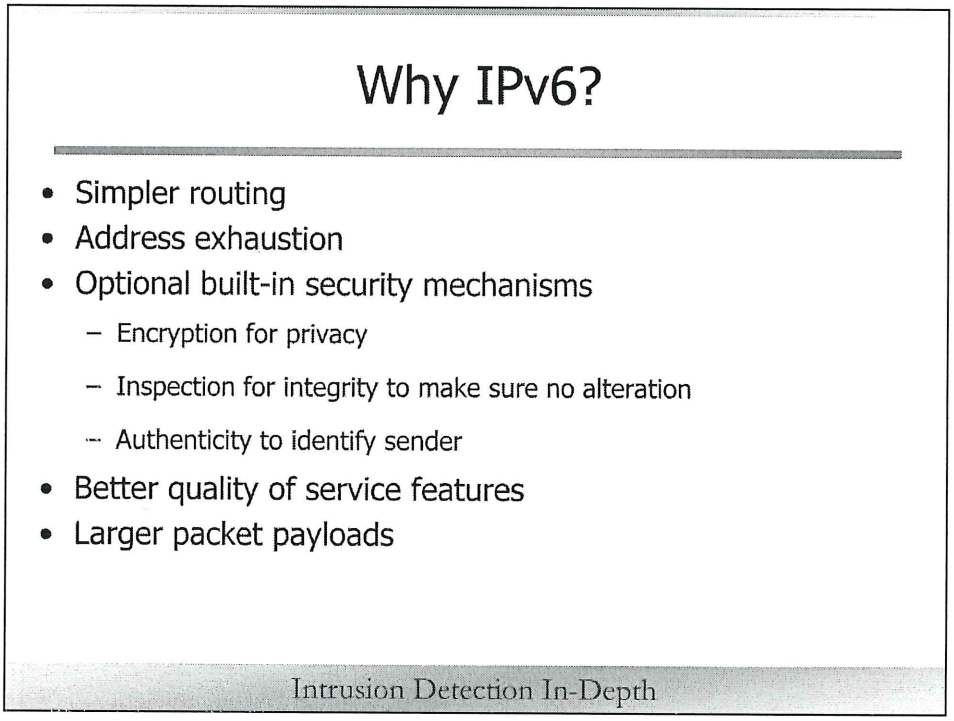
This page intentionally left blank.



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

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

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



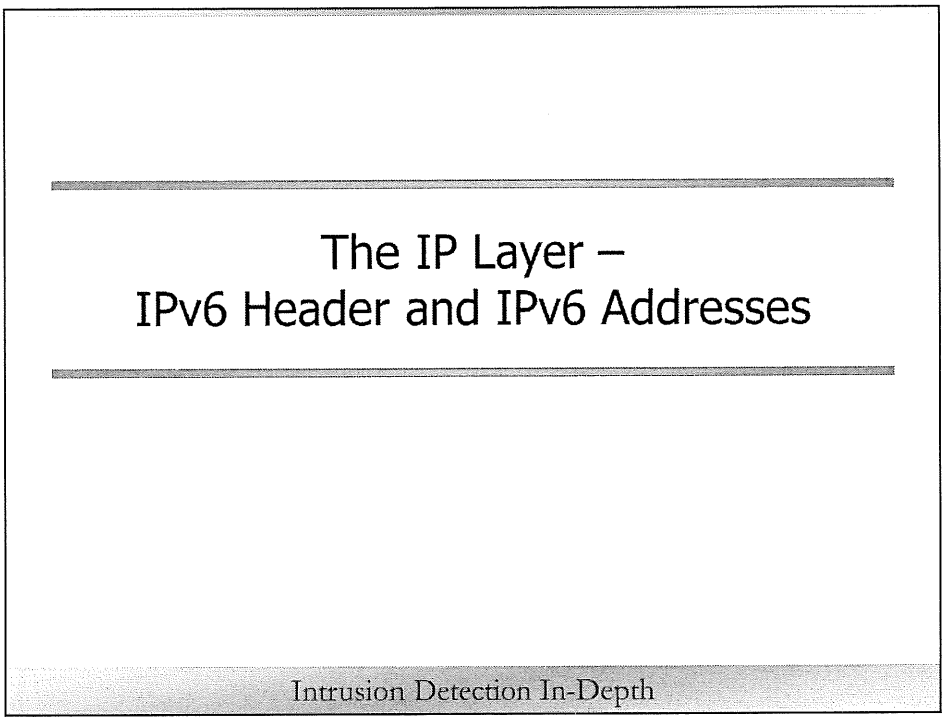
IPv4 was not created with the notion that the Internet would grow to be an international sensation complete with stationary computers, mobile devices, thieves, and miscreants. And, while it has provided adequate service for several decades, there are inherent problems that must be fixed.

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

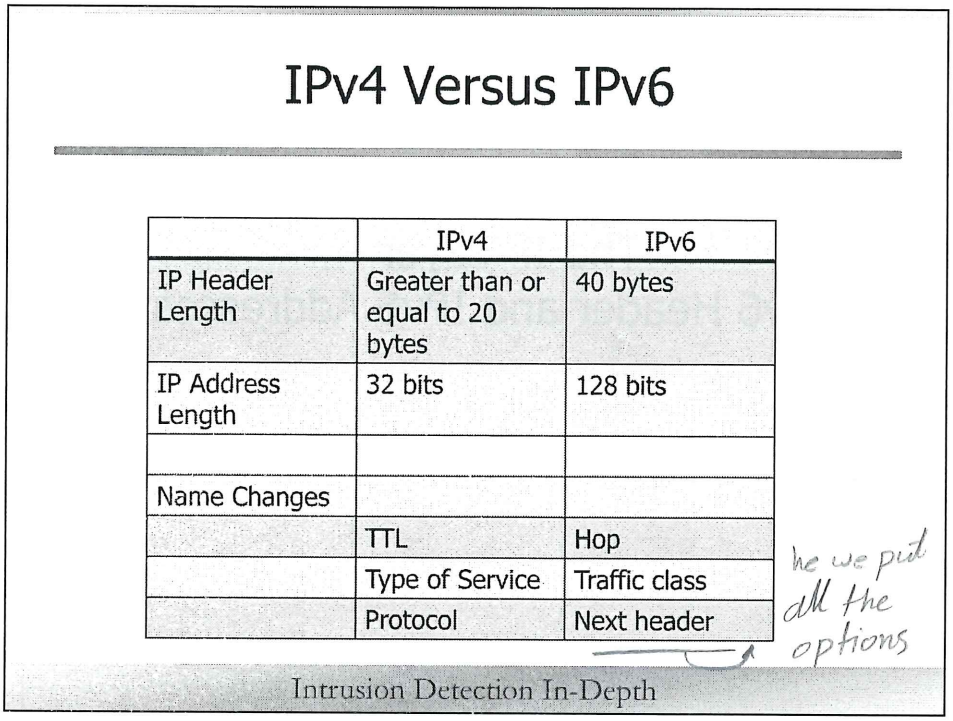
Another problem is that the 32-bit addressing scheme no longer provides enough combinations of IP addresses to accommodate the growth of the Internet and all of the many devices that now need addresses. IPv4 addresses were officially exhausted in early 2011 as IANA allocated the remaining blocks of available addresses. Although Network Address Translation (NAT) and reserved private network addresses (such as 10.0.0.0/8 and 192.168.0.0/16, etc.) have helped alleviate a more rapid exhaustion of IP addresses, a larger address space is needed. The 128-bit address scheme in IPv6 provides this relief.

And, while add-ons such as IPSec can be used for security mechanisms, they are not built into IPv4. IPv6 provides these mechanisms as extensions to a fixed-size IP header. Therefore, optional encryption and authentication can be employed to provide confidentiality and ensure the integrity and authenticity of packets without the need to retrofit protocol layering.

Better quality of service features needed for real-time applications are also available. And, now with jumbo packets, the 65,535 byte limitation on the packet size has been increased in IPv6. These are just a few of the many reasons for the creation of IPv6.



This page intentionally left blank.



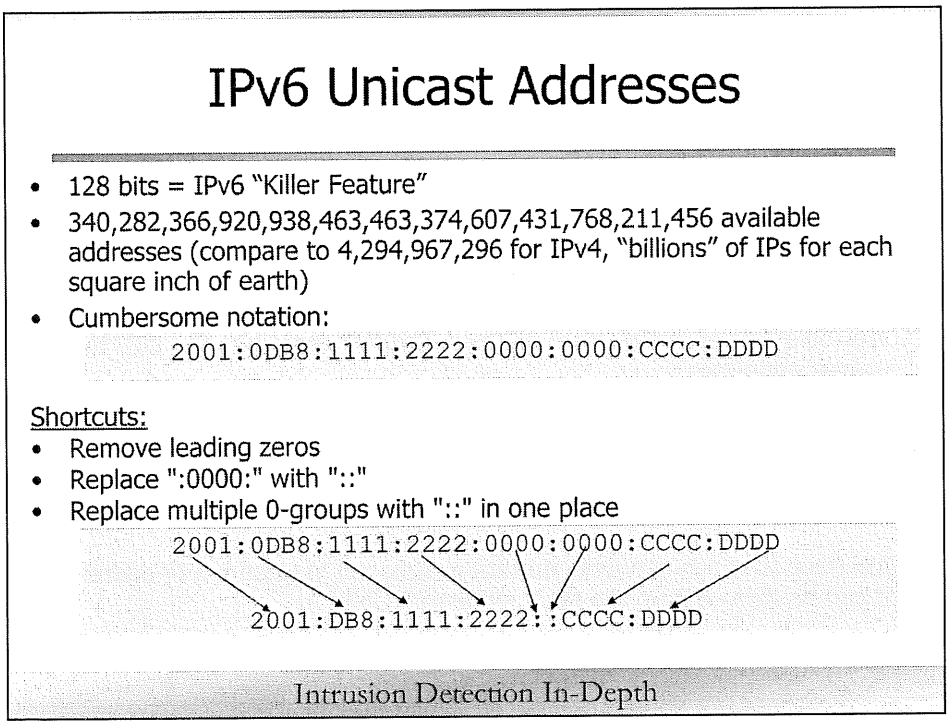
Probably the most discussed feature of IPv6 is the extended address space (more later). However, a few other things have been "fixed." The header is now limited to exactly 40 bytes. Instead, additional "extension headers" are used to express functionality covered by IPv4 fields and options. This format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

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

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

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

Fields removed: Fragmentation, Flags, Checksum, Options, IPID, Length
Fields added: Flow Label



Remembering IPv6 addresses is challenging. Typically, they are written down in 8 groups of 2 bytes in hexadecimal. A couple of shortcuts can be used. For example, leading zeros may be dropped. Groups of 0's may be replaced with "::"' However, this can be done only once per IP address in order to avoid ambiguity. Consider this example:

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

because it could either stand for:

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

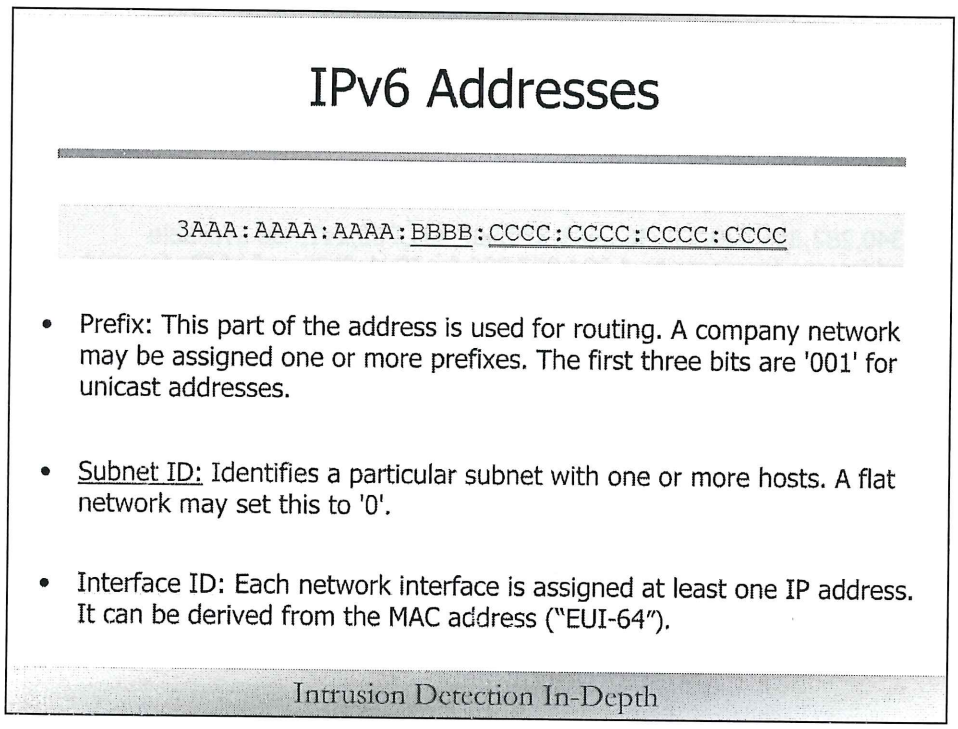
`2001:0DB8:0000:2222:3333:0000:0000:4444`

Subnets are noted like in IPv4, but the trailing 0's may be dropped. e.g.

2001:0DB8/32

Here is a quote with two analogies to illustrate the vastness of IPv6 space:

> "The available address space has been variously described in analogies from 'as many addresses as there are stars in a galaxy' or 'as many as there are grains of sand on the planet'."
>
> http://www.ip-performance.co.uk/blog/ipv6readiness



It is up to the registrar (ARIN, APNIC, RIPE...) to decide how to divide the address space. ARIN currently allocates IPv6 blocks with a minimum initial size of /32. Additional blocks are available with a size of /48. ARIN further specifies that most "sites" (e.g. a company) should be allocated a /48. A /64 is to be used if only one subnet is needed (e.g. For a consumer). (http://www.arin.net/policy/nrpm.html#six21)

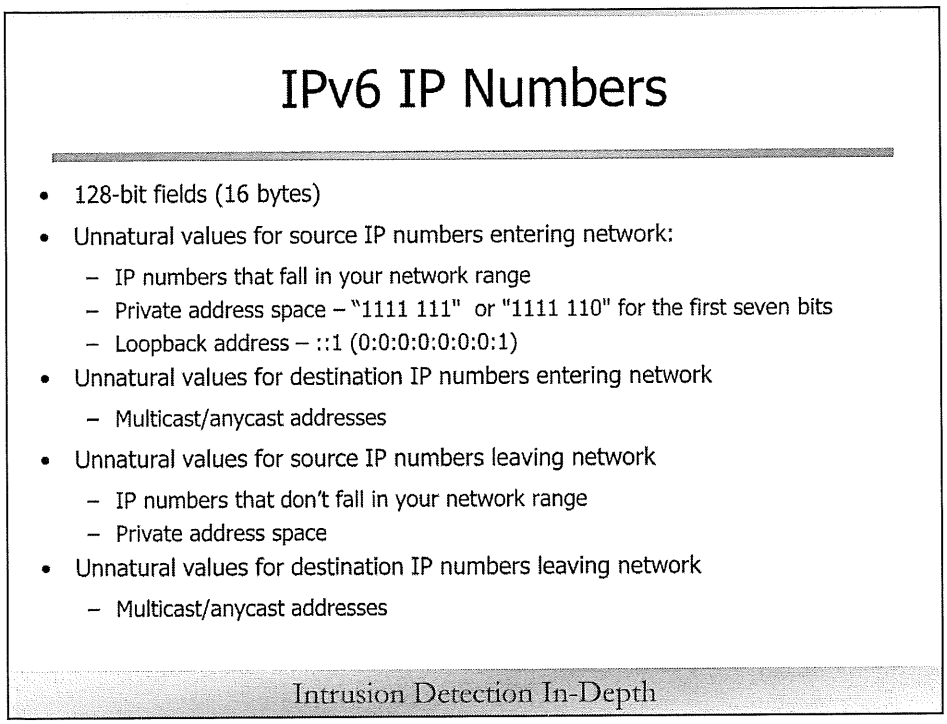
The interface ID may be derived from the MAC address using scheme known as EUI-64 format:
Use the upper 24 ("half") of the MAC address, append "FFFE", followed by the lower 24 bits of the MAC address. The upper 24 bits of the MAC address are also referred to as the Organizationally Unique Identifier (OUI) and identify the manufacturer of the equipment. The lower 24 bits should be unique for a particular manufacturer.

Sample IPv6 unicast address:

```
2AAA:AAAA:AAAA::0050:56FF:FEC0:8
```

The first nibble has to be 2 or 3 as for unicast addresses, the first 3 bits are "001".

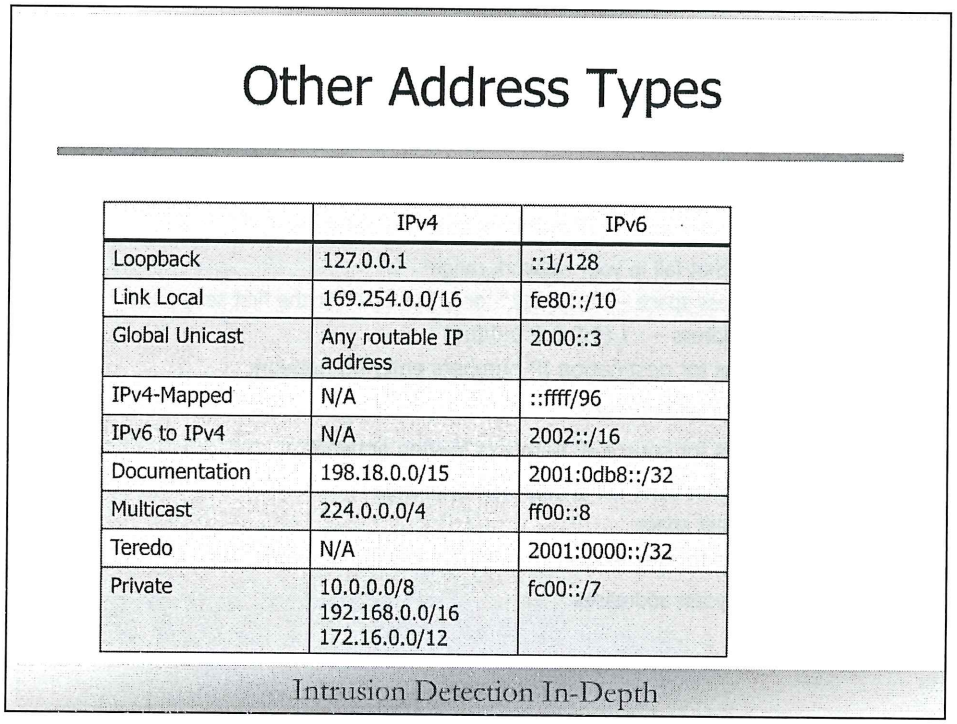
In this example, the MAC address is 00:50:56:C0:00:08. The last four digits "0008" are written as "8" according to the rule that allows us to omit leading 0's. The subnet id is "0" in this case, and compressed to "::".



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

For more information on IPv6 address assignments:

http://www.iana.org/assignments/ipv6-address-space



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

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

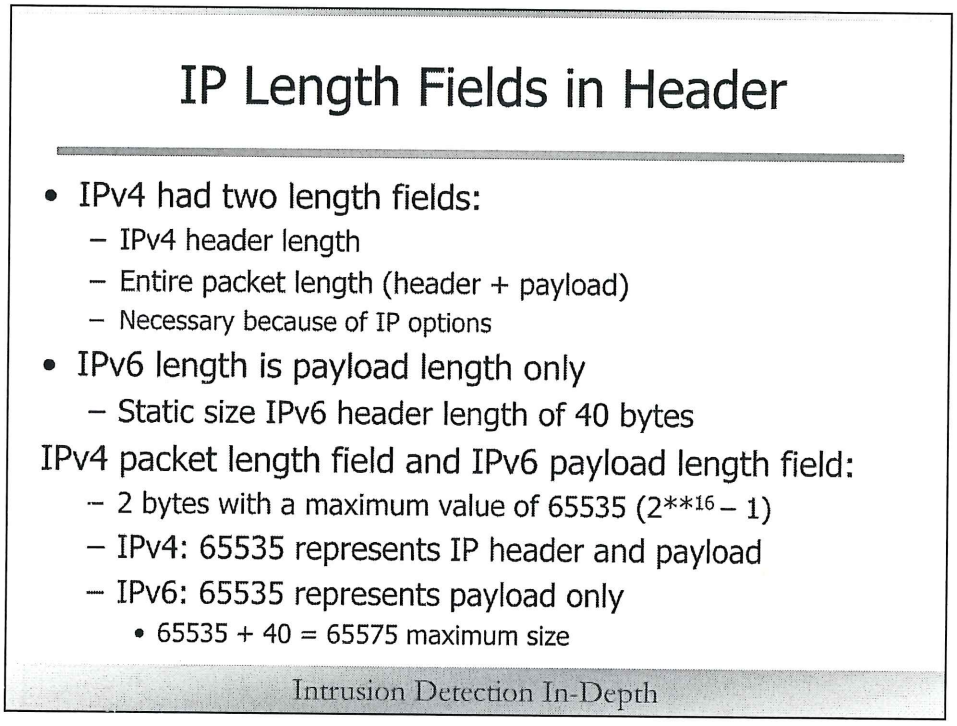
The documentation/example category is used when you want to refer to a legitimate example network. For instance, in this course any fictitious traffic that is created for the purpose of documenting should use these ranges. You are probably familiar with multicast to send traffic to a group of multicast hosts. Teredo, as you will learn, is an ill-conceived way to route IPv6 over UDP. And, the private networks are ones used locally and not intended to be routed over the Internet.

IPv6 Length Field in Header

- No more IP header length field as in IPv4
- IPv4 packet length field found in IPv4 header represented the length of the entire packet
  - Included header and payload
- IPv6 length is payload length only
  - Doesn't include the static 40 byte IPv6 header length
- IPv4 packet length field and IPv6 payload length field are both 2 bytes with a maximum value of 65535
  - IPv4: 65535 represents IP header and payload
  - IPv6: 65535 represent payload only
    - 65535 + 40 = 65575 maximum

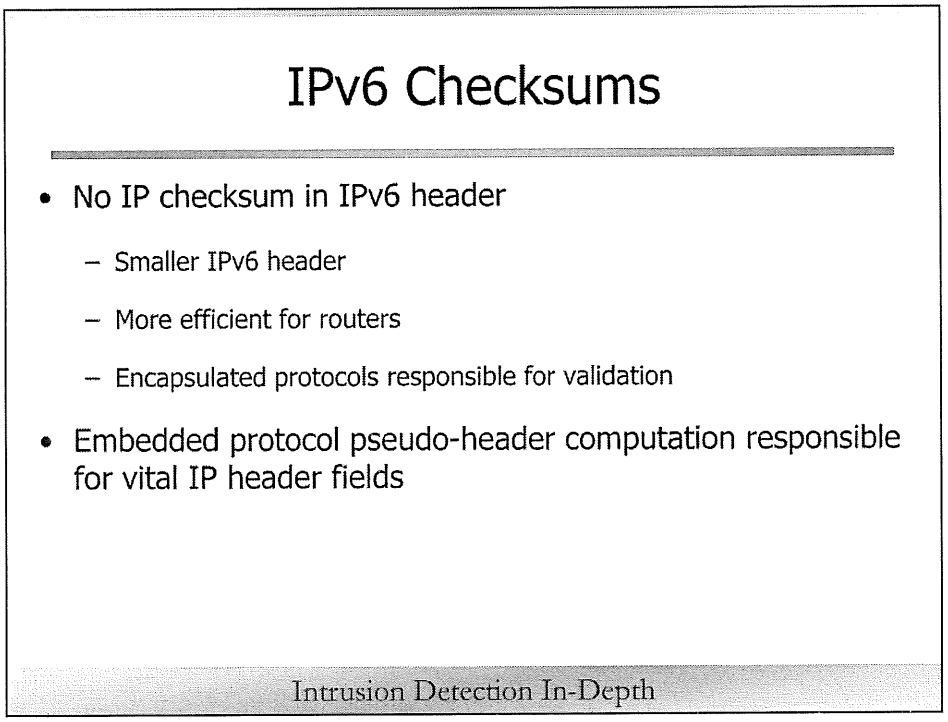
Intrusion Detection In-Depth

Used to accommodate presence of variable length IPv4 options.



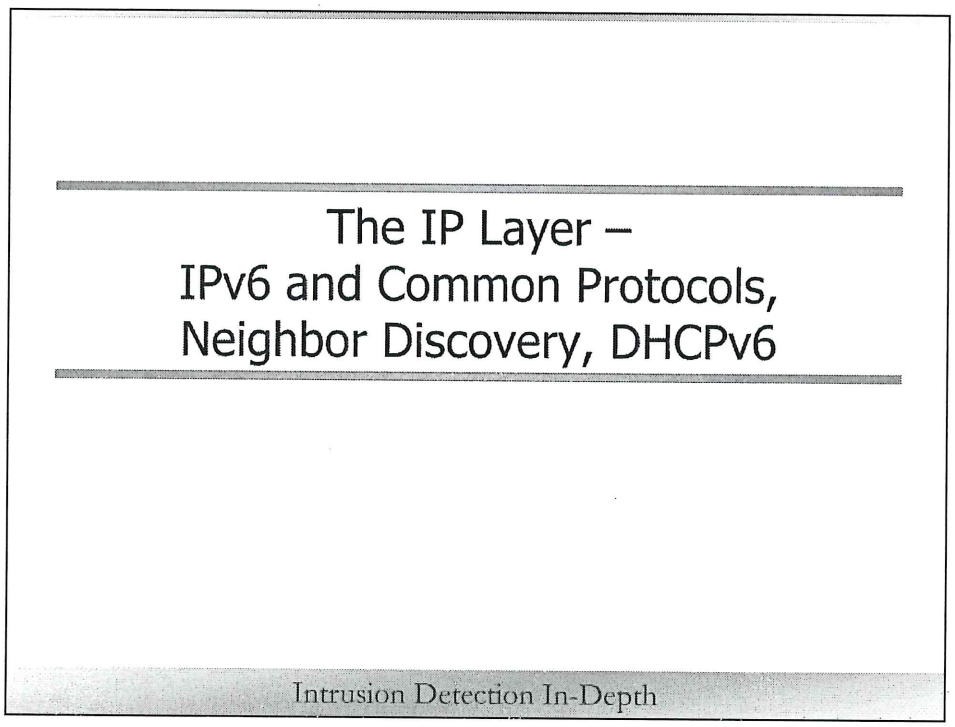
The IPv6 header has a single length field representing the size of the payload following the IPv6 header. There is no need for a field with the IPv6 header size since it is static and always 40 bytes. If you recall, it was necessary to have two length fields in IPv4 – one for the IPv4 header length and another for the total IPv4 packet length that included the IPv4 header and payload bytes. This was necessary because of the possibility of IP options that caused the IPv4 header length to be greater than the standard 20 bytes. IPv6 IP options are placed in an extension header that follows the static 40-byte IPv6 header.

Both the IPv4 total packet length and the IPv6 payload length are 2 bytes long, meaning that the maximum value that can be placed in it is 2**16 – 1 or 65535. And, that creates a limit in the number of bytes that can be placed in a packet when not including jumbograms. This means that the IPv4 total packet length is 65535. However, since the IPv6 payload length maximum is also 65535, but does not include the 40-byte IPv6 header, the maximum IPv6 packet size is 65575.

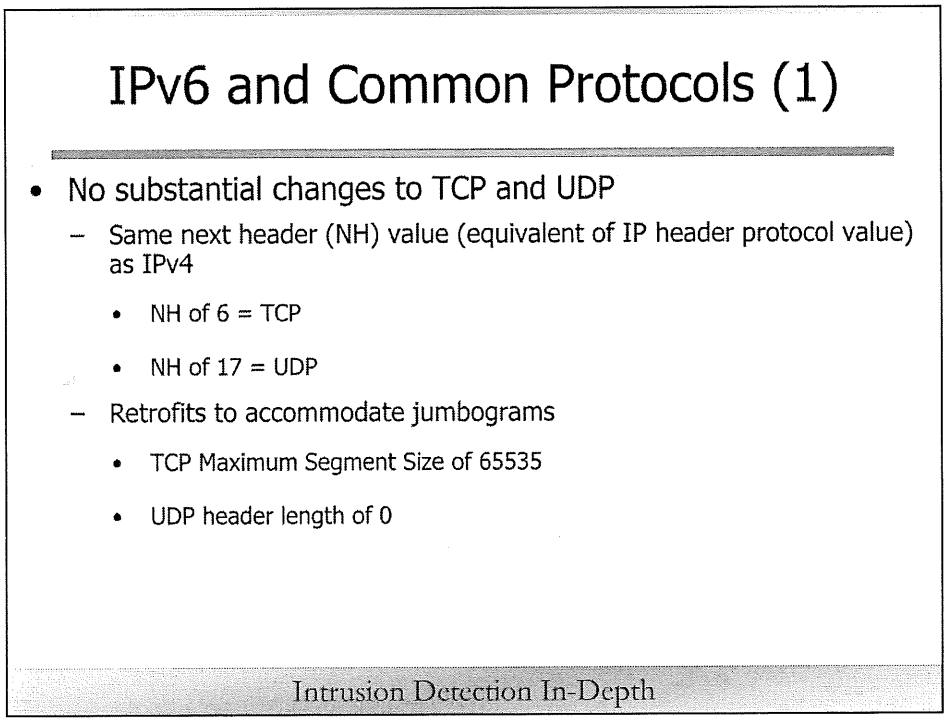


There is no longer an IP checksum associated with IPv6 traffic. This eliminates a field in the IPv6 header making it more compact. Also, there are big efficiency gains in terms of routing. If you recall, each router after decrementing the TTL had to recompute the IPv4 header checksum since a field in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

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



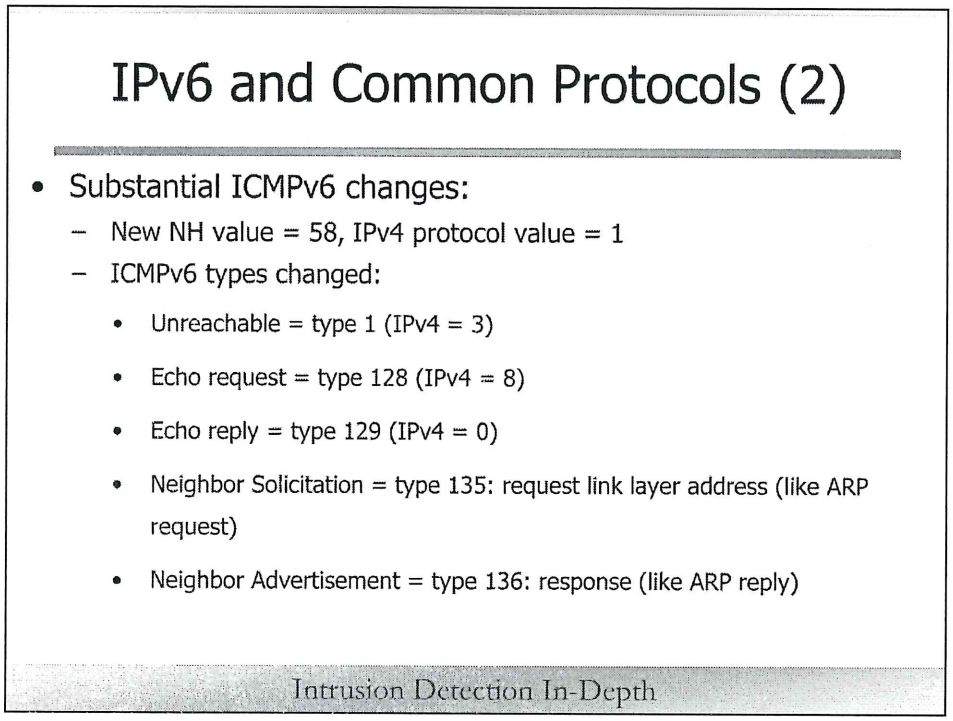
This page intentionally left blank.



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

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

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



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

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

128: Echo Request

129: Echo Reply

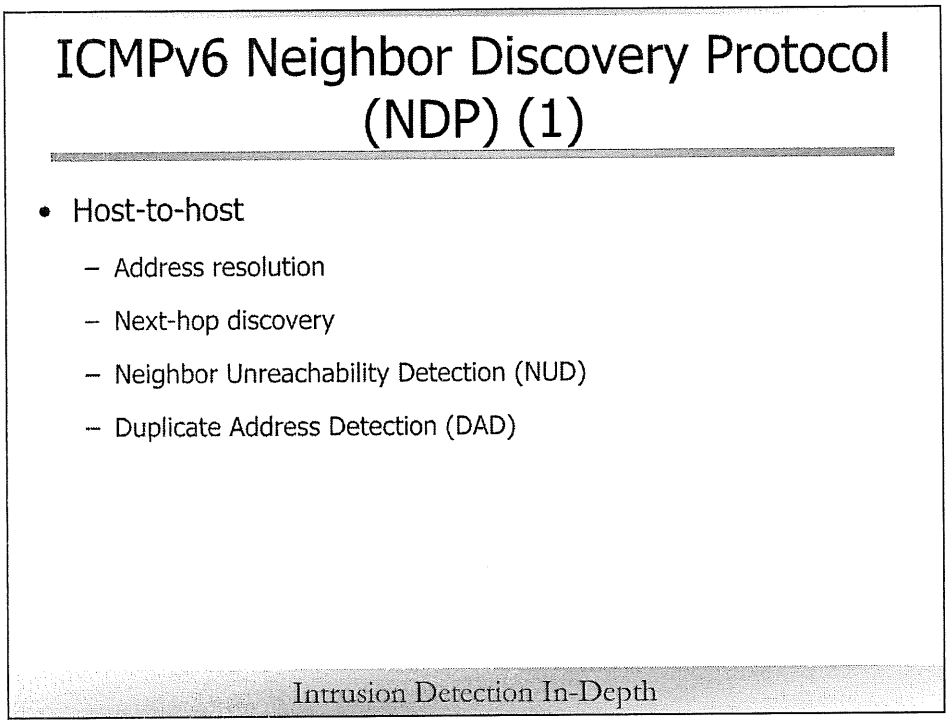
133: Router Solicitation (for auto configuration)

134: Router Advertisement (for auto configuration)

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

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

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



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

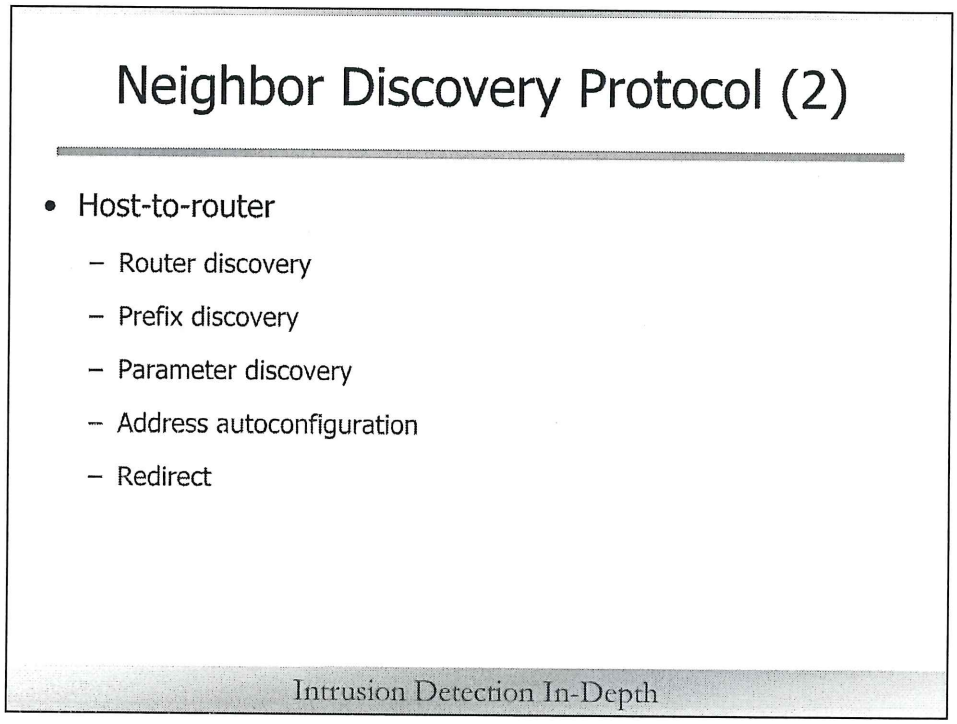
Address resolution to associate an IP and its related MAC address in IPv6 involves two different ICMPv6 exchanges – a Neighbor Solicitation - NS - (ICMPv6 type 135) sent to all hosts multicast address and a Neighbor Advertisement - NA - (type 136) from the target host to return its link layer address.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

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

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

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

If you'd like more details on NDP, look at RFC 2461 "Neighbor Discovery".

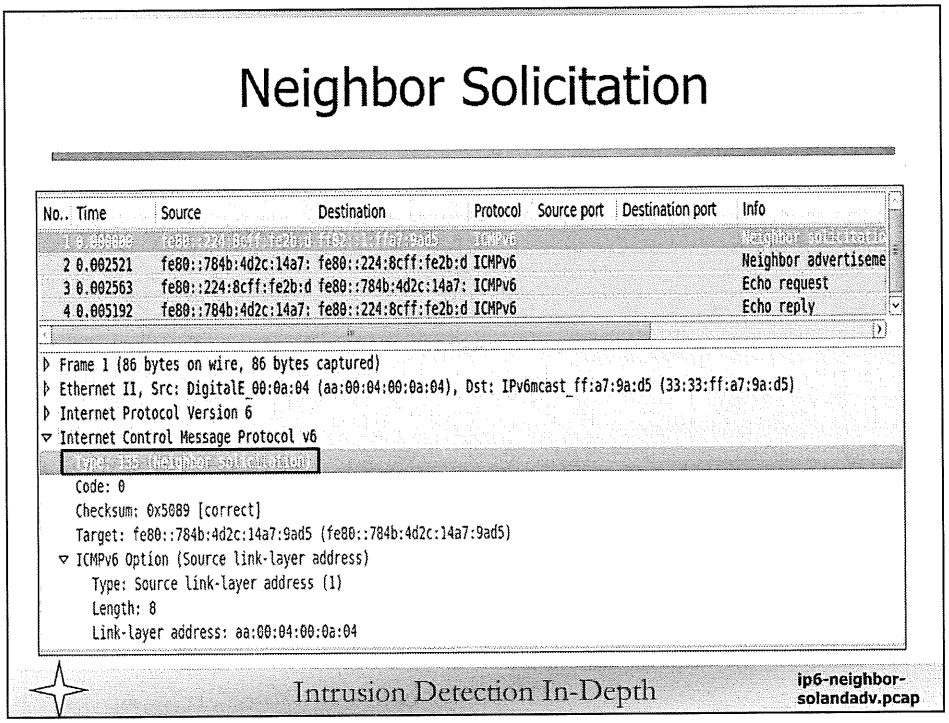


IPv4 relies on manual configuration or DHCP to find the address of the default gateway. IPv6 uses a Router Solicitation - RS - (type 133) and Router Advertisement - RA - (type 134) much like the Neighbor Solicitation/Advertisement NDP protocol. When a host first becomes part of a link, a multicast Router Solicitation is sent to the routers. Any listening router replies with a Router Advertisement sent to all hosts. This message contains the MAC address of the router, and one or more prefixes/subnets that represent the local network that do not need to be routed (Prefix Discovery), and the MTU of the link to the router along with an initial hop count value (same as IPv4 TTL) for the host to use (Parameter Discovery).

Address Autoconfiguration provides something known as "stateless" address configuration. It is called stateless because it requires no other information or interaction with any other node on the network. This may be a temporary address assigned if "stateful" configuration is later assigned by a DHCPv6 request. In between the stateless and stateful assignments, the host will use DAD to make sure there are no duplicates in the network, and make a Router Solicitation to elicit information about its network.

Address Autoconfiguration generates a link-local address. The first 10 bits of the address are "1111 1110 10", followed by 54 zeros and the lower 64 bits derived from the MAC address using EUI-64 formatting.

The NDP Redirect function is similar to the IPv4 redirect where a router informs a host of a better route. We'll discuss the concept of router redirect when we cover ICMP.



Suppose IPv6 host fe80::224:8cff:fe2b:d 64e needs to ping fe80::784b:4d2c:14a7: 9ad5 (note: the IP addresses that appear on the packet list pane are truncated by Wireshark) and has no entry in its neighbor cache where it stores IPv6 address and link address pairings. It must send a Neighbor Solicitation (NS) using an ICMPv6 message with type 135 to discover the destination host's MAC address. The destination address, in this particular case, is ff02::1:ffa7:9ad5. This requires a bit of explaining.

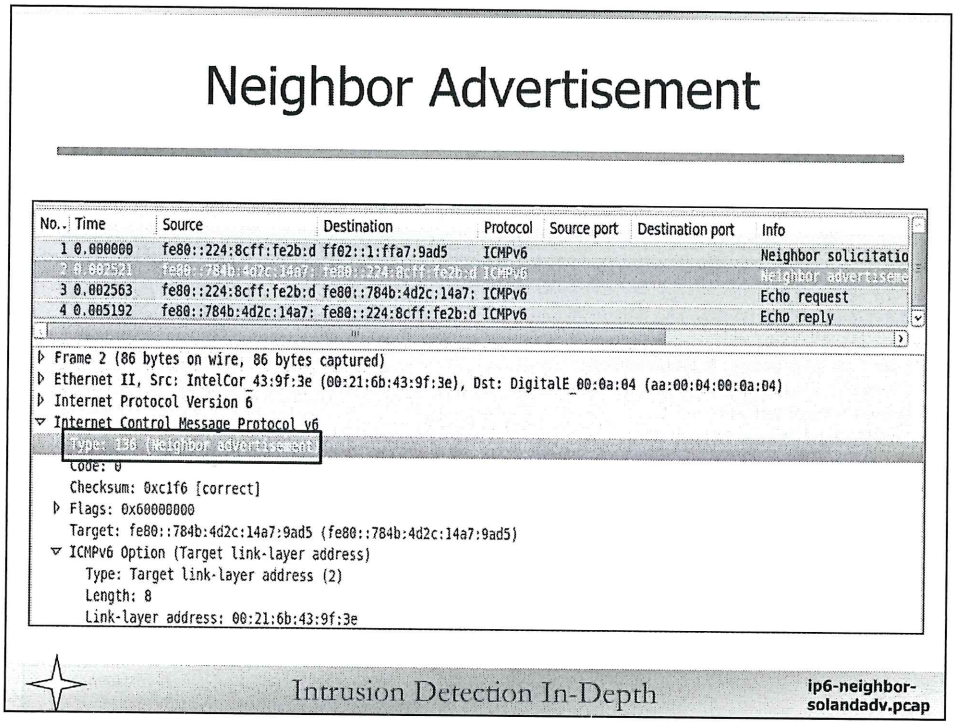
If you recall, the "ff02::1" address represents the multicast address to all hosts. A multicast address of "ff02::1:ff" is the solicited-node multicast address. It is used specifically for NDP. This prefix consumes the first 104 bits of the IPv6 address. The final 24 bits reflect the last 24 bits of the target host – in this case "ffa7:9ad5". This scheme ensures that the NS is sent to all hosts on the local network.

The source of the NS contains its MAC address as you see in the ICMPv6 source link layer option. There are many ICMPv6 options that can be associated with NDP. We'll see some additional ones in the router advertisement ICMPv6 message.

To view the output, enter at the command line:

wireshark ip6-neighbor-solandadv.pcap

This is the first record.



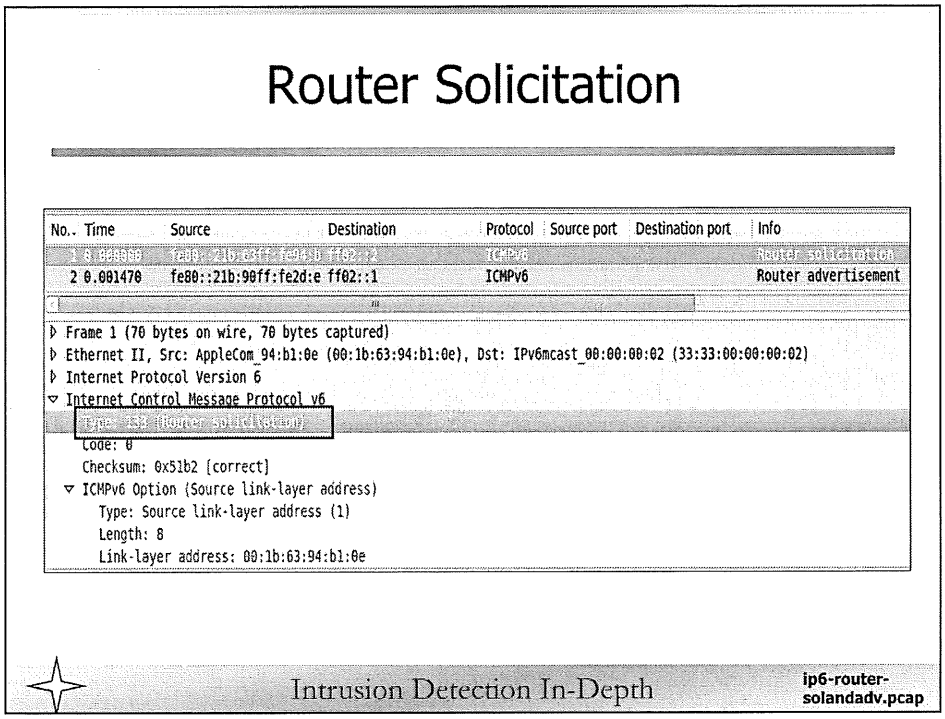
If the target host is listening, it returns a Neighbor Advertisement message using ICMPv6 type 136. Also, for some added efficiency, all neighbors that see the NS message cache the source IP and MAC address pairing for the sender of the advertisement and maintain it in their neighbor cache for a limited time. The MAC address is returned in another ICMPv6 option – the target link layer address.

The two hosts can now communicate; in this case with an ICMPv6 echo request and reply.

To view the output, enter at the command line:

wireshark ip6-neighbor-solandadv.pcap

This is the second record.

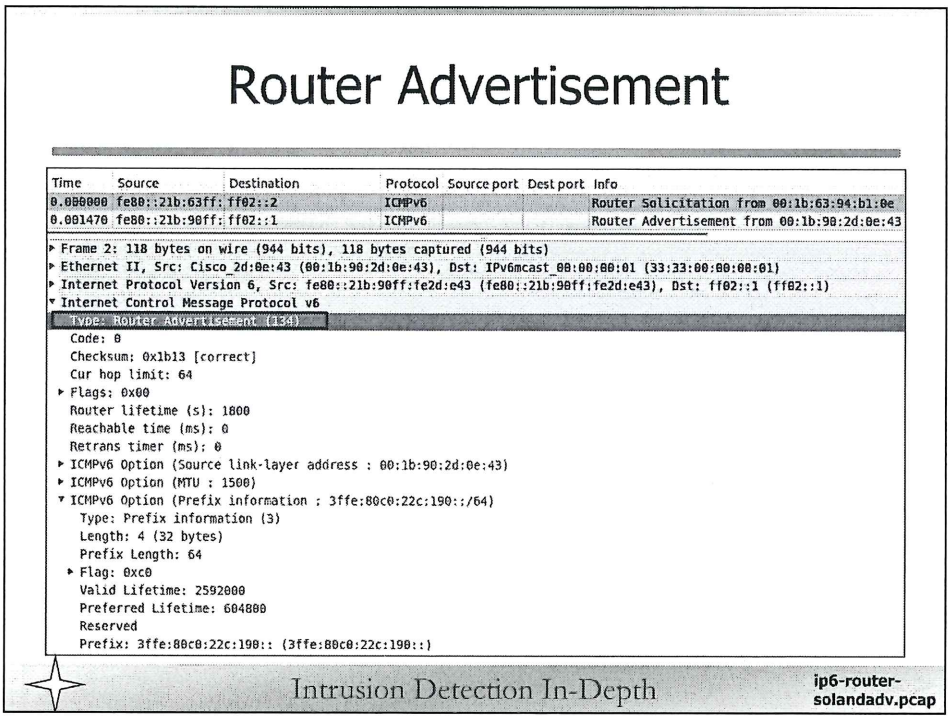


Let's examine another exchange when a host joins a network. When host fe80::21b:63ff:fe94:b10e is first connected to the network or newly rebooted, it must know its first hop router(s). It finds them by issuing a Router Solicitation message, ICMPv6 type 133, to the multicast address of ff02::2 that represents the "all routers" address. Once again you see the ICMPv6 option source link layer address with the sender's MAC address.

To view the output, enter at the command line:

wireshark ip6-router-solandadv.pcap

This is the first record.



The router with IP address of fe80::21b:90ff:fe2d:e43 responds with a Router Advertisement, ICMPv6 type 134, to destination address of ff02::1, the multicast address of all hosts. This is sent as a multicast, not unicast message so that any listening hosts may learn this same information if they do not already know it.

The router responds with some information in addition to its address. The "Cur hop limit" of 64 informs hosts that that is the hop limit value to place in the IPv6 header. If you recall, this is equivalent to the IPv4 Time to Live value that is used to designate the maximum hops/routers that a packet can traverse before being expired.

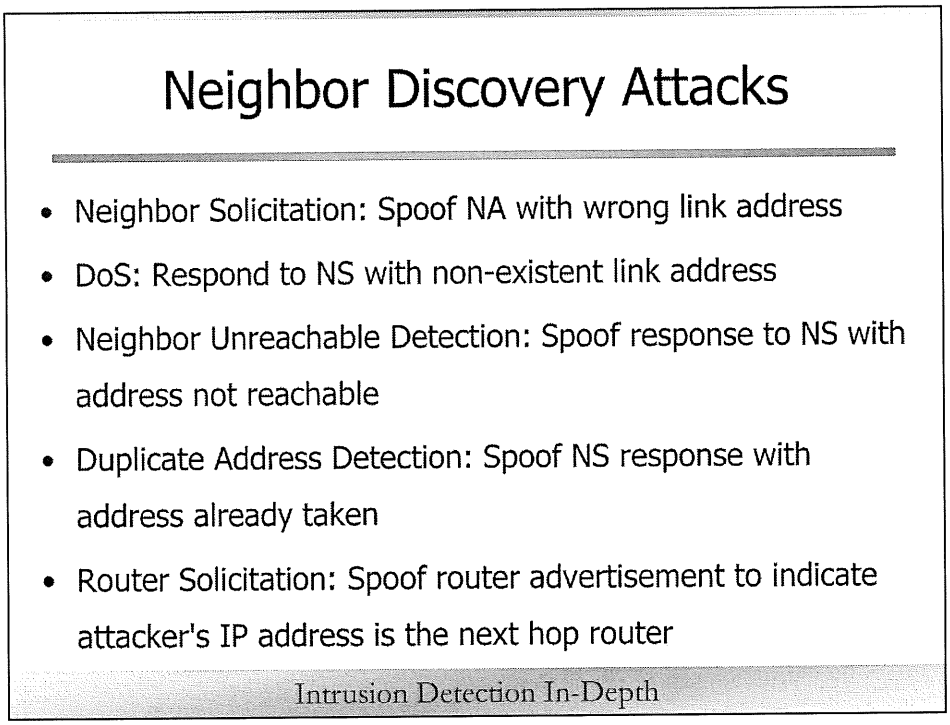
The router lifetime is the number of seconds that the information in this advertisement should be kept. This particular router information is active for 1800 seconds, after which the host(s) must make another Router Solicitation. The time information that follows informs the receiving hosts how long to consider a neighbor reachable and the time, in milliseconds, that a host needs to wait before attempting to retransmit NS messages.

There are two other ICMPv6 NDP options, the MTU and Prefix options. The MTU specifies the maximum size a packet on the network must be to avoid fragmentation. The prefix option designates the prefix to be used in the IPv6 global/routable address when stateless address autoconfiguration is used. If you recall there are several different IPv6 type addresses for a given interface. The link-local ones begin with fe80:: and are used for local/non-routable traffic. The prefix is applied to the global/routable IPv6 address for the interface. In this case the prefix is the 64 high-order bits of 3ffe:80c0.22c:190::/64.

To view the output, enter at the command line:

wireshark ip6-router-solandadv.pcap

This is the second record.



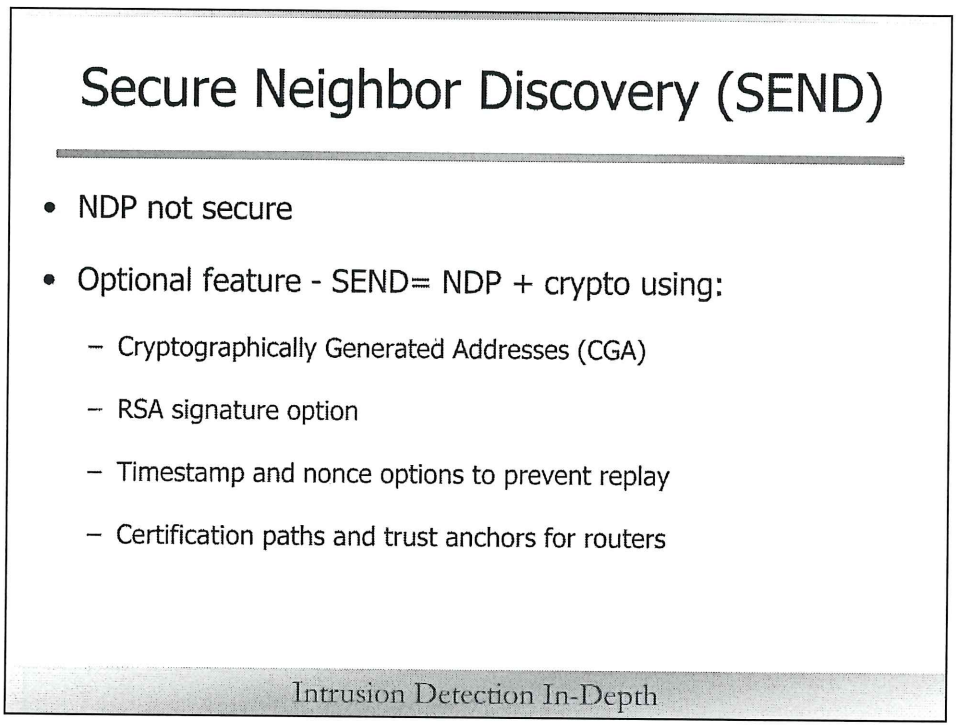
Much like the issues with IPv4 ARP, and ICMPv4 router messages, the presence of an attacker on an IPv6 network can disrupt and redirect legitimate traffic. This is easily done using NDP messages since there is no inherent security assuring that the message is not spoofed by a malicious node.

An incorrect IPv6 host address/link layer pairing can be performed in one of two ways. The first and more obvious way is for the malicious host to respond to a Neighbor Solicitation (NS) request, spoofing a bogus address in the returned Neighbor Advertisement (NA). This was known as ARP poisoning in IPv4. A second way is to send an incorrect IPv6 host address/link layer pairing in the NS request itself since neighbors seeing this will place the pairing in their neighbor cache. This is similar to the use of a spoofed gratuitous ARP in IPv4. Another attack, a DoS, can be performed when the NS request receives a spoofed NA response indicating that the target host is at a non-existent link address, denying communication with the target host.

An analogous attack is the use of the Neighbor Unreachable Detection to return a spoofed NA response that the target host is unreachable. If you recall, a new node joining a network sends a NS message to determine if other hosts have a duplicate address. A DoS can be attempted if the malicious host can spoof a response indicating that the address is already in use via the Duplicate Address Detection. If the originating host attempted to generate one or more new addresses, the malicious host could respond again with DAD, effectively denying the host an address and the ability to communicate on the network.

Finally, when a host joining a network makes a Router Solicitation to discover its router(s), a spoofed Router Advertisement could indicate that the attacker's host is the router, thereby routing all traffic through it, creating a man-in-the-middle attack.

These are just a few of the many NDP attacks that can be performed. If you are interested in reading about others and proposed fixes, see RFC3756 "IPv6 NDP Trust Models and Threats".



As you have just learned, NDP is not secure from attackers inside a network. In order to secure NDP messages, the source of the message must be associated with the actual sender – not spoofed. This can be done using an optional feature known as SEcure Neighbor Discovery (SEND). SEND assures the integrity of the message to prevent against manipulation. SEND employs a timestamp to preclude replay attacks and a nonce to match an advertisement with a solicitation. The NDP message has different options fields that follow the NDP data to carry all optional data associated with SEND options.

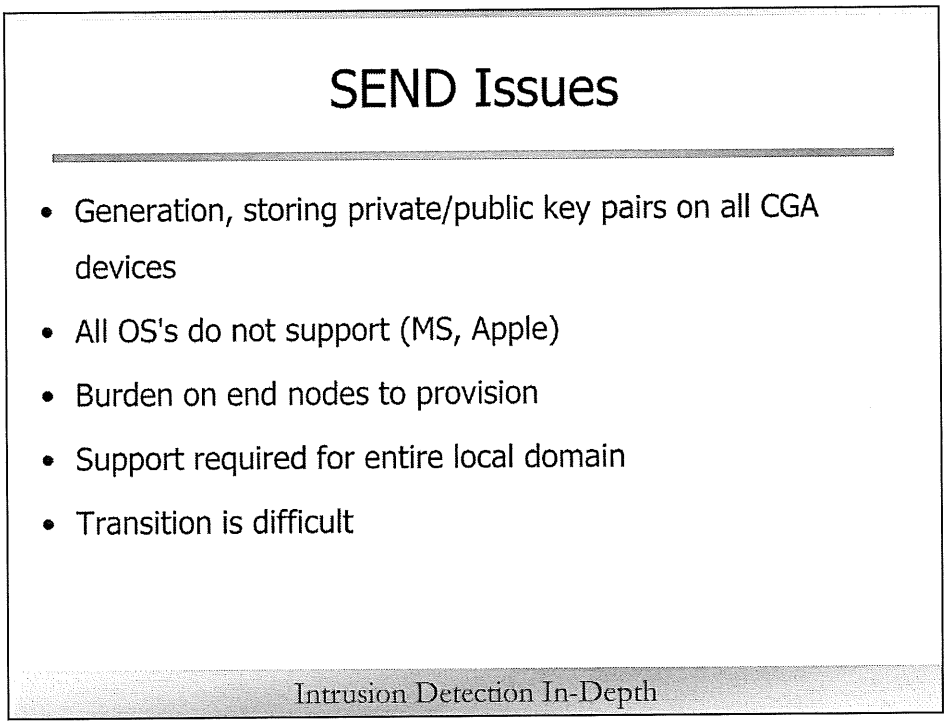
The first step in the process of securing NDP is for each node to have a Cryptographically Generated Addresses (CGA) public-private key pair before claiming an address. The CGA validates the sender is authentic. The public key and associated data are placed in the CGA option of the NDP message.

Optionally, messages can be signed with the public key using a hash of the IPv6 and ICMPv6 headers, and the NDP message, and all NDP options that precede the signature option. A signed message protects against manipulation in transit. This information is placed in the RSA option.

The timestamp option places a timestamp value in an advertisement to make sure it is not replayed later. A nonce is a one-time randomly generated value by the sender and placed in the nonce option of a solicitation message. A valid associated advertisement will have the same nonce.

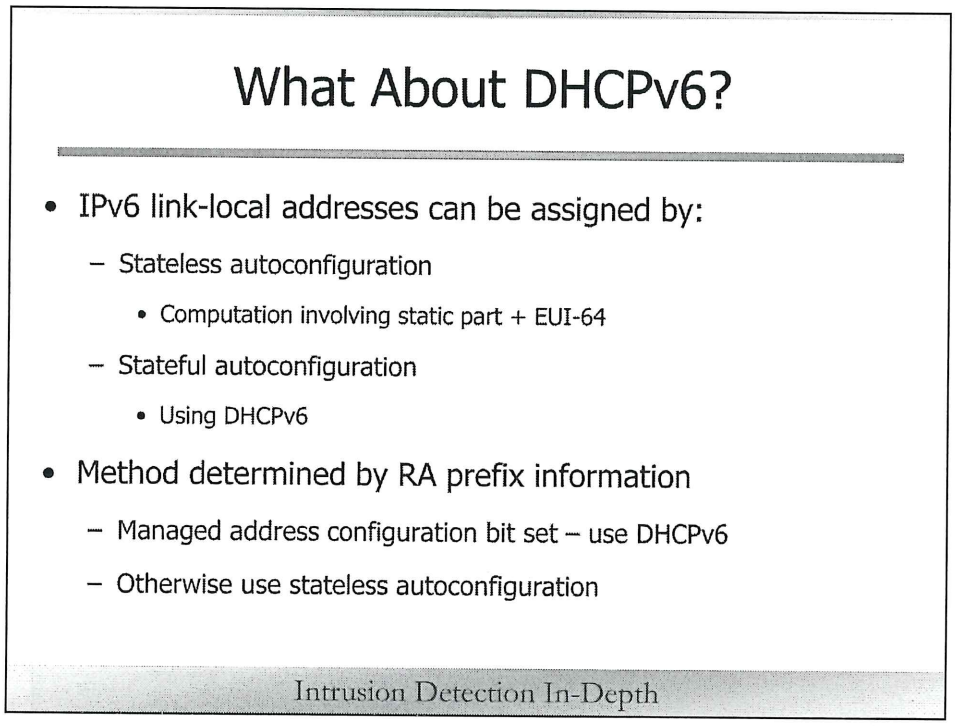
There are additional optional protections to ensure that routers are trusted. Each node is configured with one or more trust anchors that are used to authorize routers. Trust anchors can be defined locally or globally and are akin to certificate authorities. When a given node receives a message from a router, it has to establish what is known as a "certification path" to the trust anchor for the router. The trust anchor authorizes the router.

More information on SEND can be found in RFC 3971 "SEcure Neighbor Discovery".



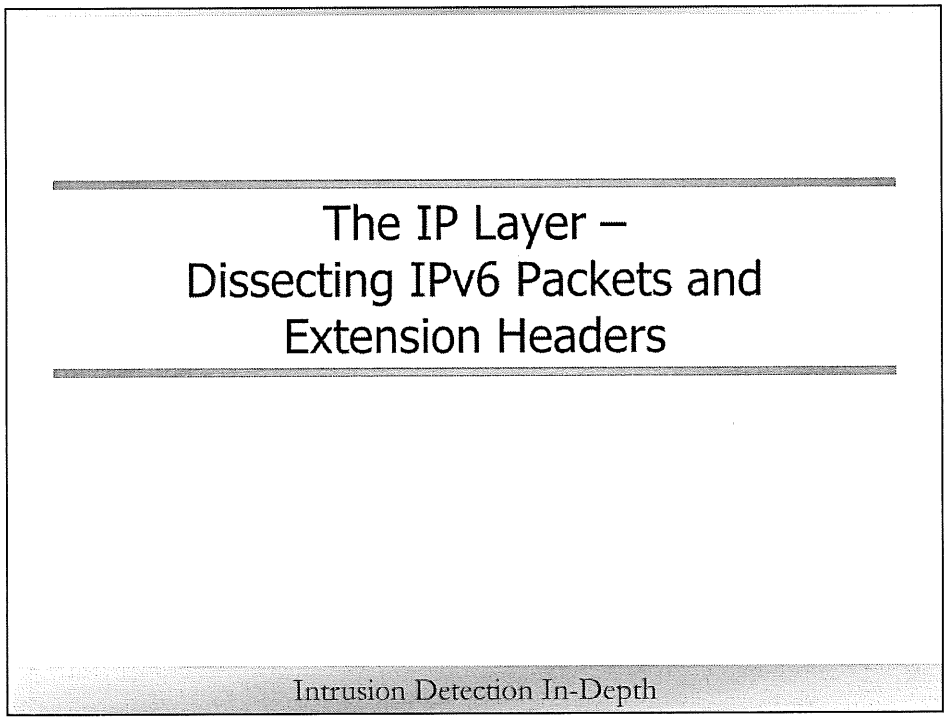
There are many issues that may keep SEND from being a viable implemented protection mechanism. Any node has to be able to generate and permanently store its own private and public key. How this is done and secured is different from device to device – routers, and different operating systems. Further, SEND requires support for these new NDP options from all nodes on a network to make it secure. This means routers as well as host operating systems. Currently, neither Microsoft nor Apple provides SEND functionality.

All end hosts and routers have computational burdens of verifying the sender's address and possibly the entire NDP message if the RSA signature option is used. The use of timestamps means that hosts have to be provisioned with synchronized times, most likely using Network Time Protocol (NTP). SEND requires support from the entire local domain to completely assure that NDP messages are valid. As is apparent, the transition to SEND may be painful or non-existent due to all the work required and breadth of functionality that needs to be supported.

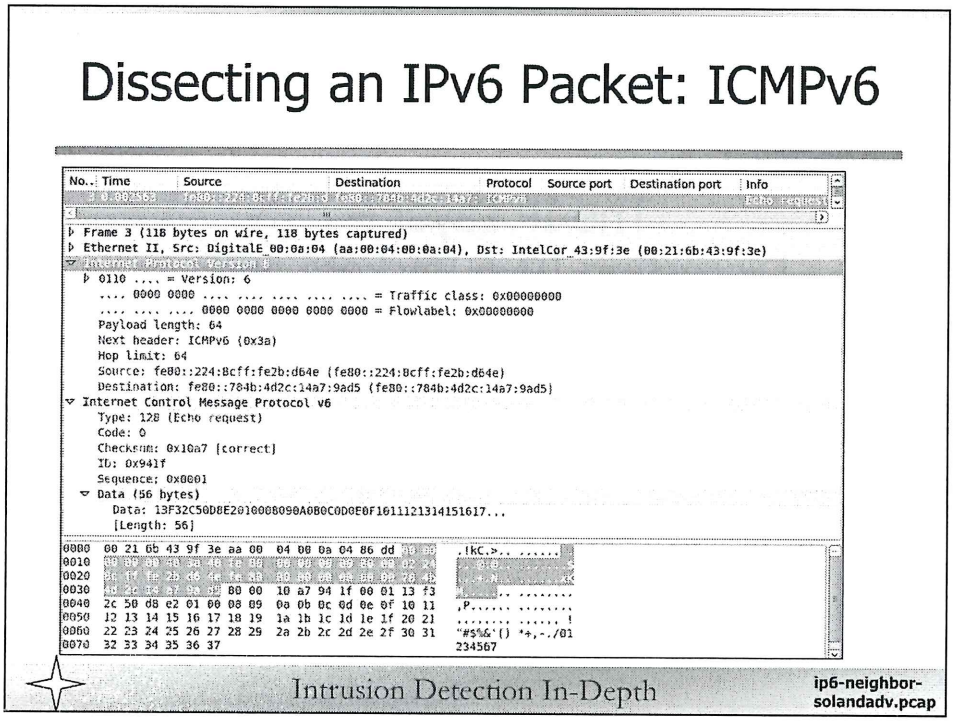


As you have just learned, the first way to assign an address is via Stateless Address Autoconfiguration to generate a link-local address. The first 10 bits of the address are "1111 1110 10", followed by 54 zeros and the lower 64-bits derived from the MAC address using EUI-64 formatting.

Another option, in addition to or in lieu of Stateless Address Autoconfiguration, is Stateful Address Autoconfiguration using DHCPv6. How does a given node know whether to use DHCPv6 or not? If you remember, when a node joins a local network, it issues a Router Solicitation. The Router Advertisement returns information about the local router(s) and contains a bit known as the managed address configuration bit. DHCPv6 is used if this bit is set. Obviously, the router must be configured so that this bit is properly set in the Router Advertisement message.



This page intentionally left blank.



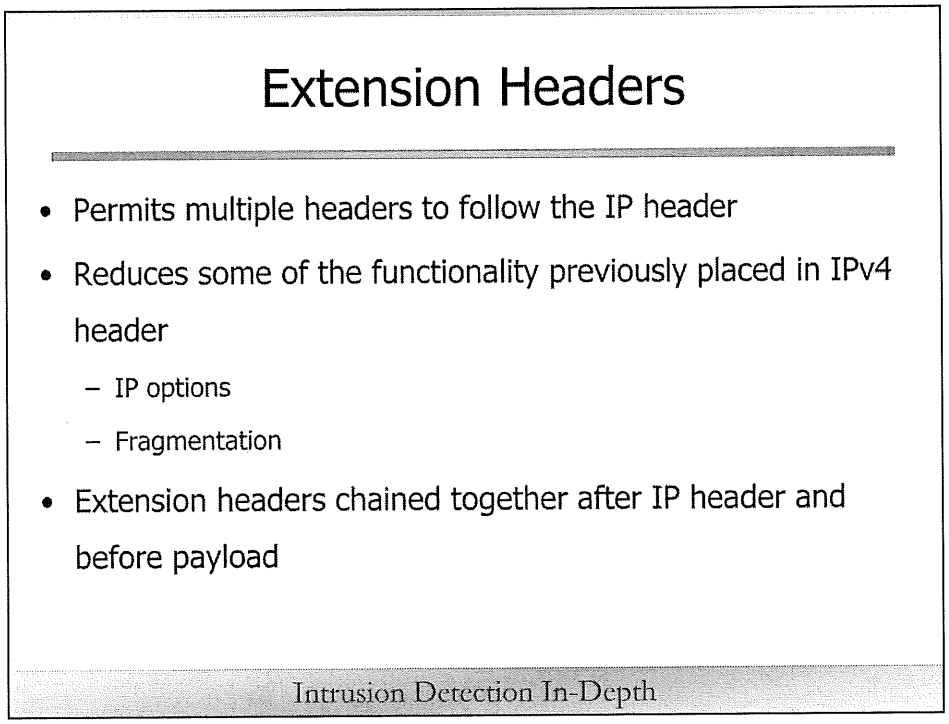
Let's take a look at the format of an IPv6 packet. ICMPv6 is a good protocol to examine since ICMP has changed from IPv4. This is an ICMPv6 echo request to show you how IPv6 looks in hex. The IPv6 header is visible at the bottom in the bytes pane highlighted in blue. The ICMPv6 header follows with underlined bytes. The remaining portion is data. The IPv6 header is a fixed 40 bytes. So there is no need to give the header length as required in IPv4. Any additional IPv6 header fields/data are placed in something known as an extension header that we'll soon discuss.

The next header field in the IPv6 header is the equivalent of the IPv4 header protocol. ICMPv6 has a new protocol/next header value of 0x3a or 58. The ICMPv6 header has a different type for an echo request – 0x80 or 128 versus a type of 8 for IPv4.

To view the output, enter at the command line:

wireshark ip6-neighbor-solandadv.pcap

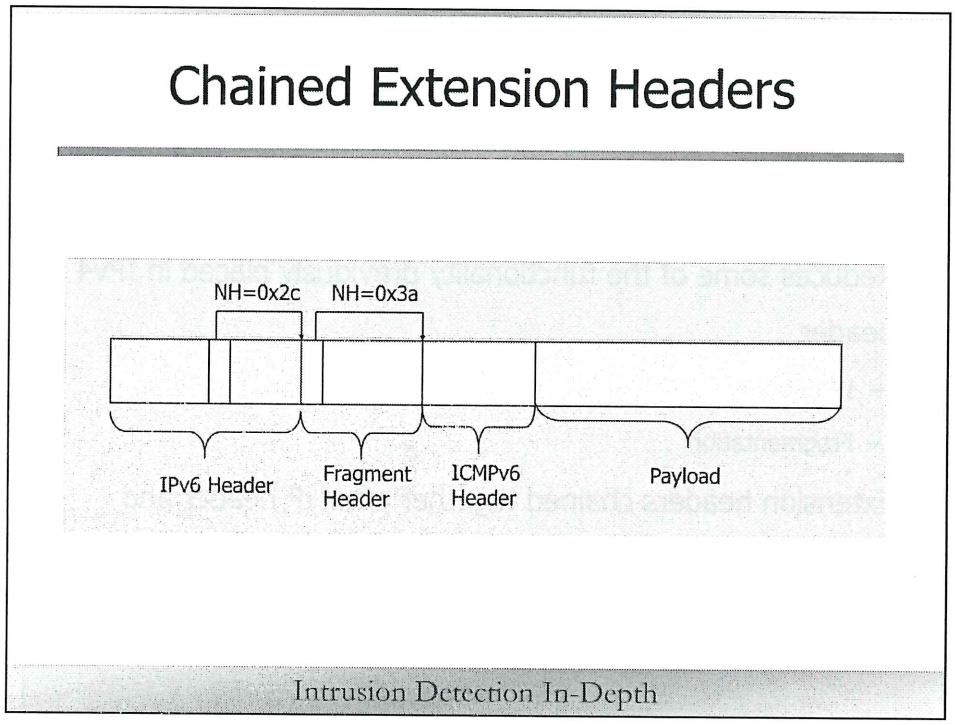
This is the third record.



While IPv4 allowed some encapsulation of protocols such as Authentication Header and Encapsulating Security Protocol after the IP header, IPv6 has expanded on the notion of providing one or more extension headers after the IP header. This removes some of the fields and functionality once found in the IPv4 header and simplifies the IPv6 header. Theoretically, the use and size of IPv6 extension headers is limited only by the maximum size of the IP datagram.

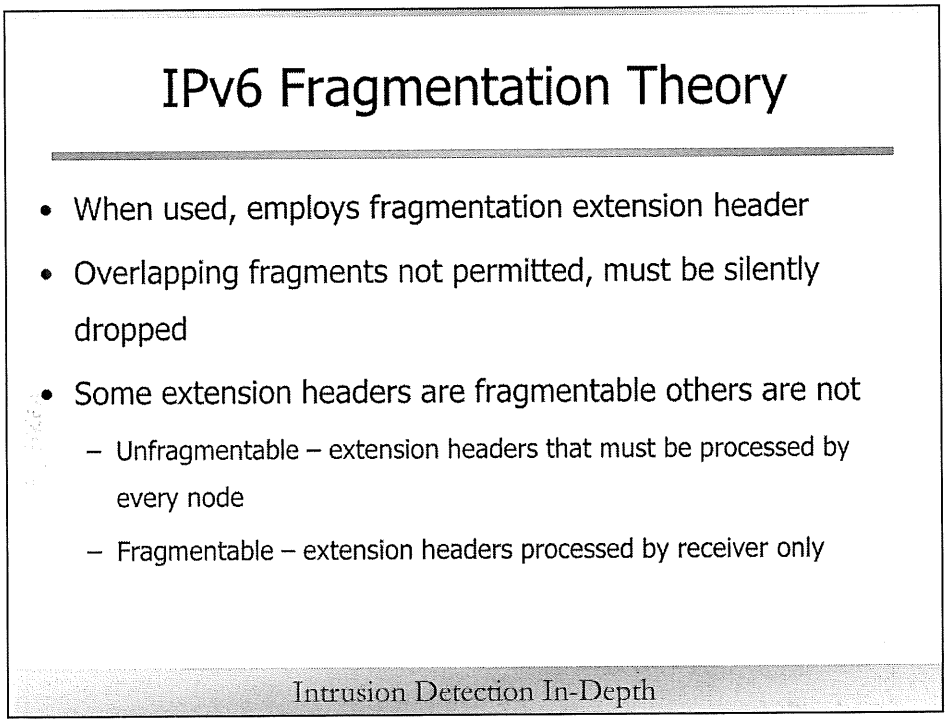
The chaining of IPv6 extension headers is accomplished by using a field called the "next header" that contains the value of the protocol that follows the current one. This is analogous to the IPv4 protocol field that contained the value of the protocol following the header.

Perhaps you are wondering about some kind of required order for extension headers if you are using multiple extension headers. RFC 2460 offers guidance that there are certain extension options, such as hop-by-hop, destination, routing and fragment headers that should be in a specified order and before any of the other extension headers. Take a look at the RFC if you are interested in the details.



This is a simple depiction of the use of chained extension headers. As mentioned before, the fragmentation fields in IPv4 – fragment offset, fragment ID, and the more fragments flag have been removed from the IP header and placed in a fragment extension header. The IPv6 header in this slide has a next header field and it contains the value 0x2c (decimal 44) indicating that the fragment header follows.

The first field in every extension header is the next header and it contains the value of the header following it. If no extension header follows, it contains a value of 59 or "no next header." In our example, the fragment header next header has a value of 0x3a (decimal 58), indicating that the ICMPv6 header follows it. From here, things look more like IPv4 where the ICMP header contains the ICMP type and code of the payload that follows.

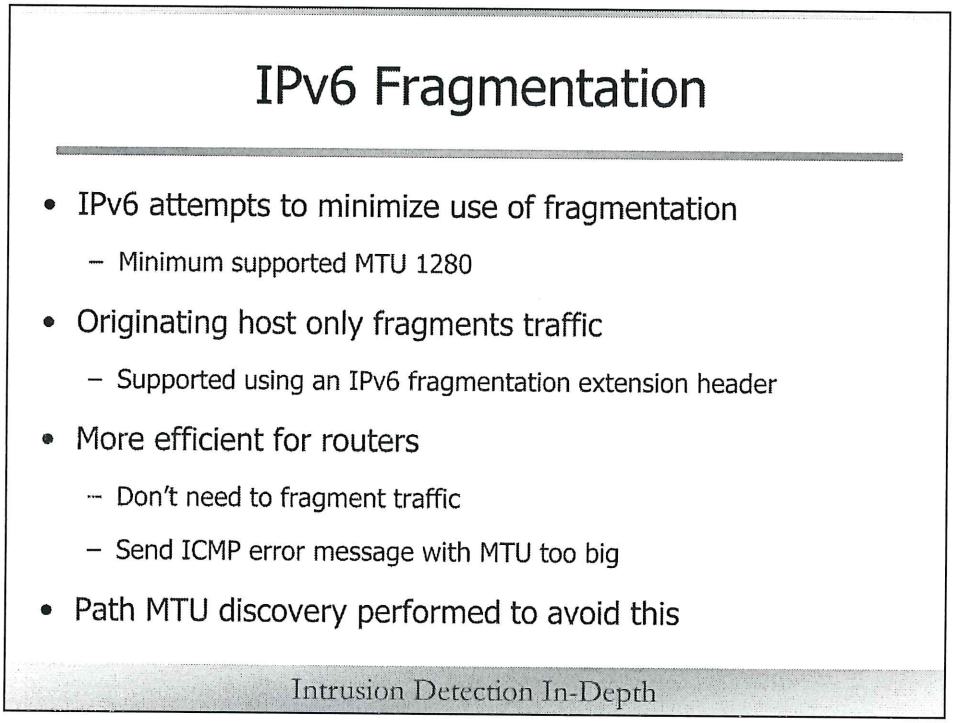


The use of fragmentation is discouraged in IPv6. However, when it is used, the fragment fields – offset, unique identification number, and whether or not more fragments follow – are placed in something known as a fragmentation extension header. This will be covered in more detail in the next set of slides, but it is basically a header expressly for fragmentation that is placed somewhere after the IPv6 header and before any protocol headers or payload.

Overlapping fragments – ones that profess to be located at overlapping offsets – are silently dropped by the receiving or reassembling host. Overlapping fragments are most likely an evasion technique and have no known benevolent purpose.

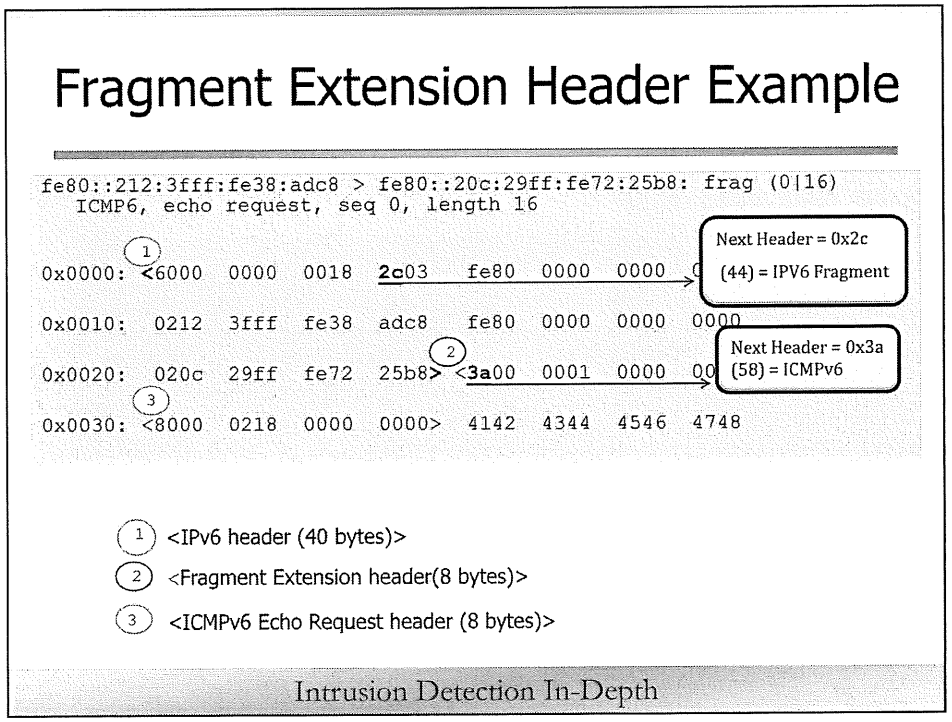
Fragmentation of an IPv4 packet begins after the IP header – in other words the IP header cannot be fragmented. The notion is expanded in IPv6 to include not only the IPv6 header but any extension header, such as hop-by-hop or routing, that must be processed by the sender, intermediate devices, and the receiver. All extension headers that are processed by the receiving host only may be fragmented.

RFC 2460 "Internet Protocol, Version 6(IPv6) Specification"
RFC 5722 "Handling of Overlapping IPv6 Fragments"



Fragmentation introduces a lot of inefficiencies. In IPv6 this isn't as burdensome since routers no longer fragment packets – the sending host only fragments packets. Still, it is best avoided and IPv6 attempts to do just that.

IPv6 supports fragmentation if absolutely necessary but discourages its use in a couple of ways. The first is that the minimum MTU of an IPv6 link is 1280 bytes, large enough for most packets. Next, there is a the Path MTU discovery where a host attempts to send a packet. If the packet is too large for a given link, an ICMPv6 "Packet Too Big" message is returned with the MTU size of the intermediate link. The sending host should reduce the packet size so that it can be wholly contained in the returned MTU size.



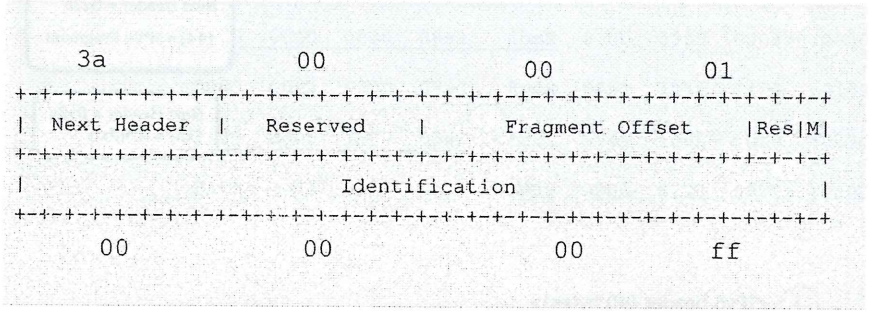
Let's look at a 0-offset fragment in hexadecimal. Each different header in the dump has been delimited with the less than "<" and greater than ">" symbols. The first header is the IPv6 header containing a next header of 0x2c. This means that a fragment extension header follows it.

The fragment extension header is 8 bytes long. If we examine the fragment extension header in more detail, we see that the first byte contains the next header value of 0x3a (ICMPv6 follows). The second byte (0x00) is a reserved field. The next 13 bits are for the fragment offset value, followed by 2 reserved bits, followed by one bit for the more fragment flag (0x1) and the last 4 bytes are the fragment id (0x0000 00ff). This isn't much different from the fields used IPv4 fragment handling. The only difference is that the fragment ID is now 4 bytes where it used to be the 2-byte IP identification field.

The final header is the ICMPv6 header. It has a type of 0x80 that says that it is an ICMPv6 echo request. The last part of the packet is the ICMPv6 echo request payload.

Fragment Extension Header Format

3a00 0001 0000 00ff

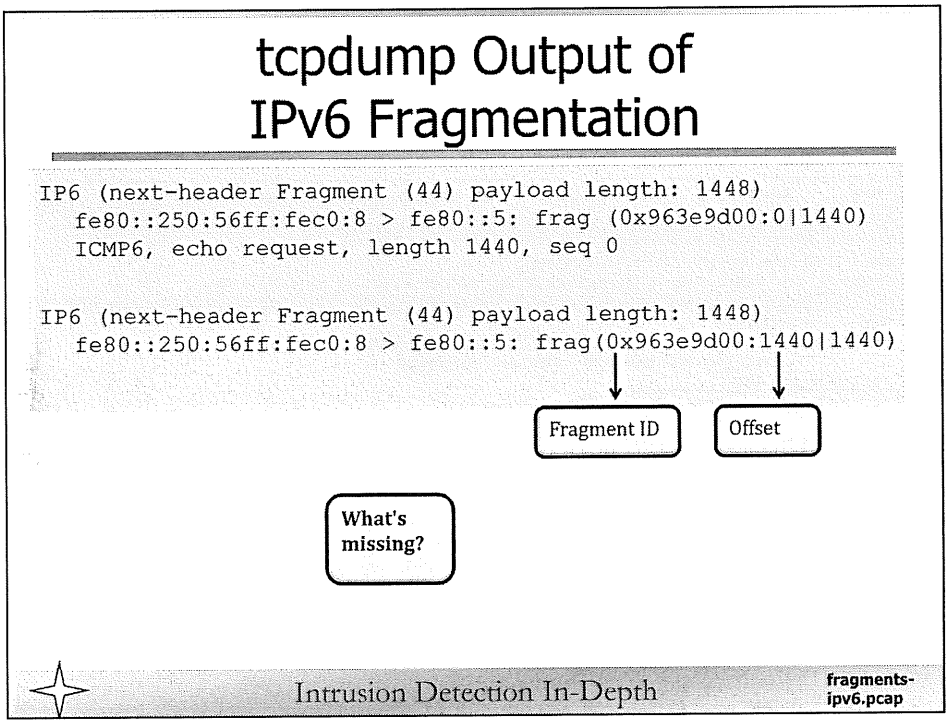


Intrusion Detection In-Depth

As noted, in the previous slide, the IPv6 extension header is not much different than the fields associated with fragmentation in the IPv4 header. There is a "next header" field in the fragmentation extension header that indicates what type of header or protocol follows. And, the fragmentation identification that used to be a 16-bit IP identification is a 32-bit value.

The way fragments are reassembled does not change. All fragments with similar source and destination IP addresses, the same protocol, and fragment identification numbers are considered part of the same fragment train. The offset and whether or not the more fragments flag is set provide the fragment's position in the fragment train and indicate whether or not more fragments follow.

You can read more about the fragmentation extension header and other extension headers in RFC 2460.
RFC 2460 "Internet Protocol, Version 6(IPv6) Specification."

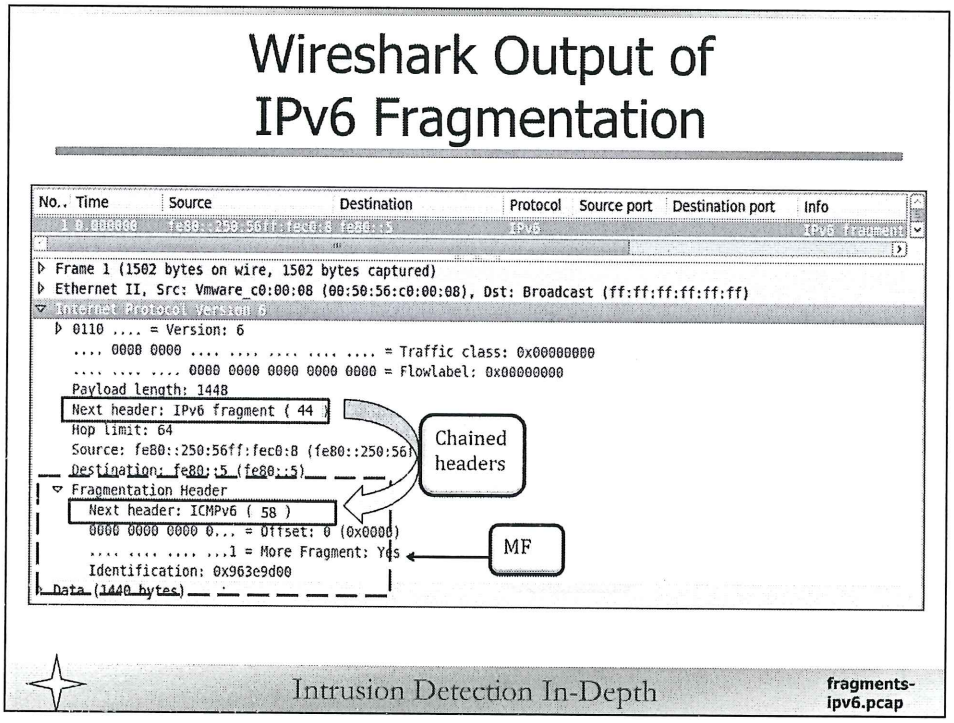


The tcpdump capture of a fragmented ICMPv6 echo request is shown above. There are two related IPv6 fragments.

Let's examine tcpdump's translation of the first one. Verbose mode causes tcpdump to indicate that there is an extension header – namely a fragment extension header with a next header value of 44. This value is placed in the next header of the IPv6 header. The fragment ID is a 32-bit value of 0x963e9d00. We see that this is the first fragment since the offset is 0 and the payload length is 1440. **The value of the MF flag is missing** from the tcpdump output, preventing us from knowing whether or not there is a last fragment. We'll examine the fragment extension header in Wireshark in the next slide where we'll see that there is still the MF flag.

The second fragment differs only in that the offset is 1440. As noted, tcpdump does not give us an indication of whether or not the MF flag is set.

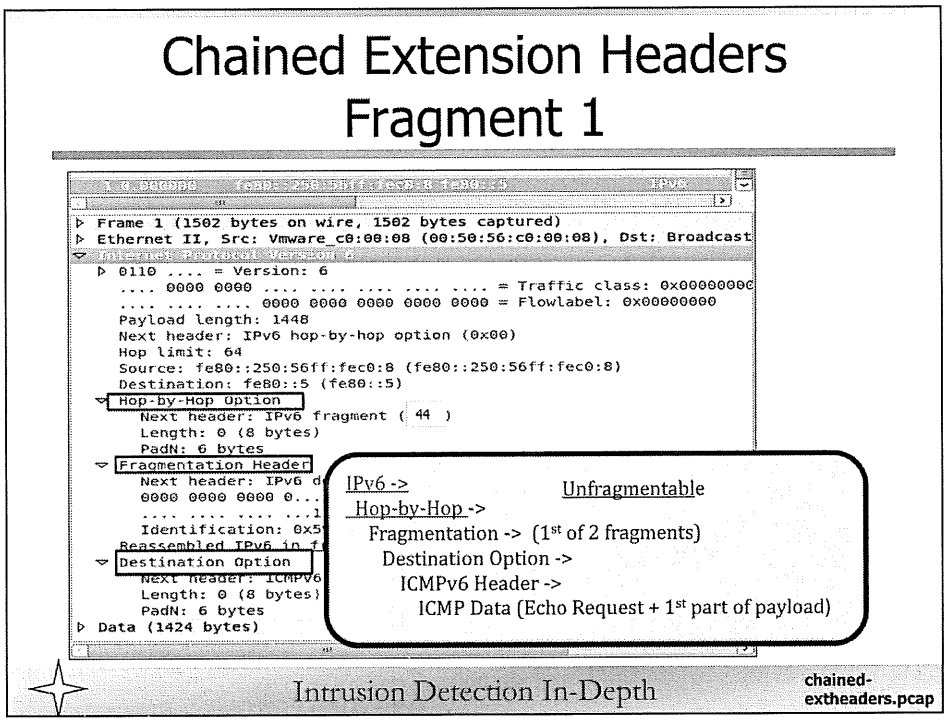
To view the output, enter in the command line:
tcpdump –r fragments-ipv6.pcap -nvt



This is the first fragment in Wireshark. It presents a more comprehensive and coherent view of the format of the fragment. The IPv6 next header is the fragment header. The fragment header has a next header of ICMPv6. Let's look at the fragment header. It retains the 13-bit field format for the offset. The MF flag still exists regardless of tcpdump's failure to place it in the output. And the fragment ID has gone from 2 bytes in IPv4 to 4 bytes in IPv6.

To view the output, enter in the command line:

wireshark fragments-ipv6.pcap



Let's examine the notion of the unfragmentable versus fragmentable extension headers using Wireshark to convey the pertinent concepts of IPv6 and extension headers.

The first extension header in use here is the hop-by-hop option that must be processed by all nodes including the sender, receiver, and all those in between. One of the goals of IPv6 is to improve the efficiency of routing packets. If an extension header such as the hop-by-hop option needs to be examined by all routing devices, it stands to reason that it should be unfragmented since reassembly of fragments is time-consuming and inefficient.

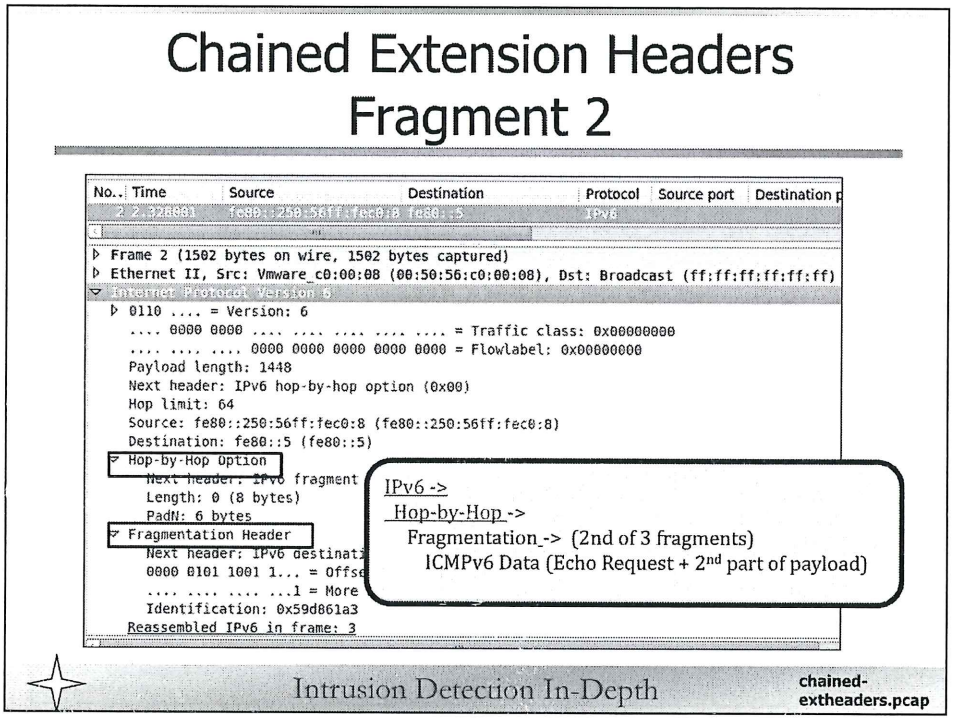
Other options that need to be processed by the destination node only can be fragmented. In this example, as the name suggests, the destination option extension header is used by the receiver only. Since the receiver must reassemble the fragments anyway, this extension header can be fragmented.

The Wireshark output display shows the first of three fragments that comprise an ICMPv6 echo request. The first layer is the IPv6 header that, like the IPv4 header is unfragmentable; it has a next header value for the hop-by-hop extension header – again not fragmentable – and necessarily must precede the fragment extension header. It has a next header value of the fragment extension header. The fragment extension header follows and has a next header value of the destination option extension header. The first fragment itself contains the destination option extension header, the ICMPv6 echo request header and the first part of the ICMPv6 payload.

To see the output, enter in the command line:

wireshark chained-extheaders.pcap

This is the first record.



Let's look at the second of three fragments. As expected, it contains an identical IPv6 header. Also, it must contain the same hop-by-hop extension header, since it is unfragmentable and must be duplicated in every fragment. The IPv6 header has a next header value for this hop-by-hop extension header.

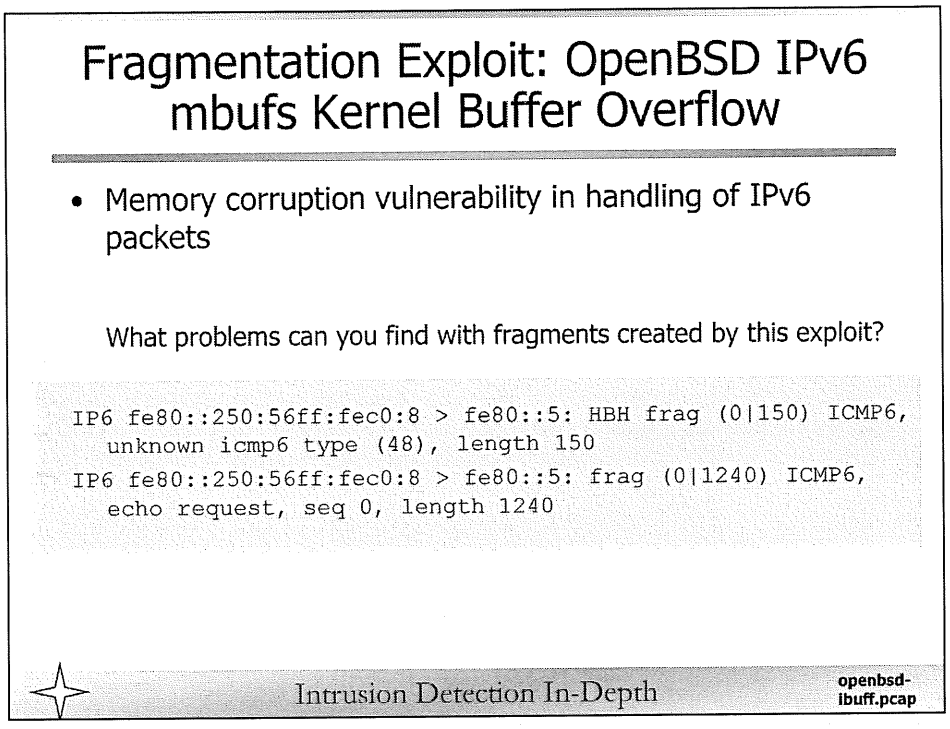
The hop-by-hop extension header contains a next header option for the fragment extension header. ICMPv6 payload follows. If you recall from our study of IPv4 fragmentation, only the offset 0 fragment has all protocol headers that follow. All remaining fragments contain an IP header – IPv4 or IPv6 and more payload data. The destination option is no longer found in this fragment. Since it is fragmentable, it was associated with the first fragment only as part of the extension header.

Neither the destination option nor the ICMPv6 echo request header is found in the second fragment since they were wholly contained in the first fragment. Therefore, only payload follows the headers.

To see the output, enter in the command line:

wireshark chained-extheaders.pcap

This is the second record.

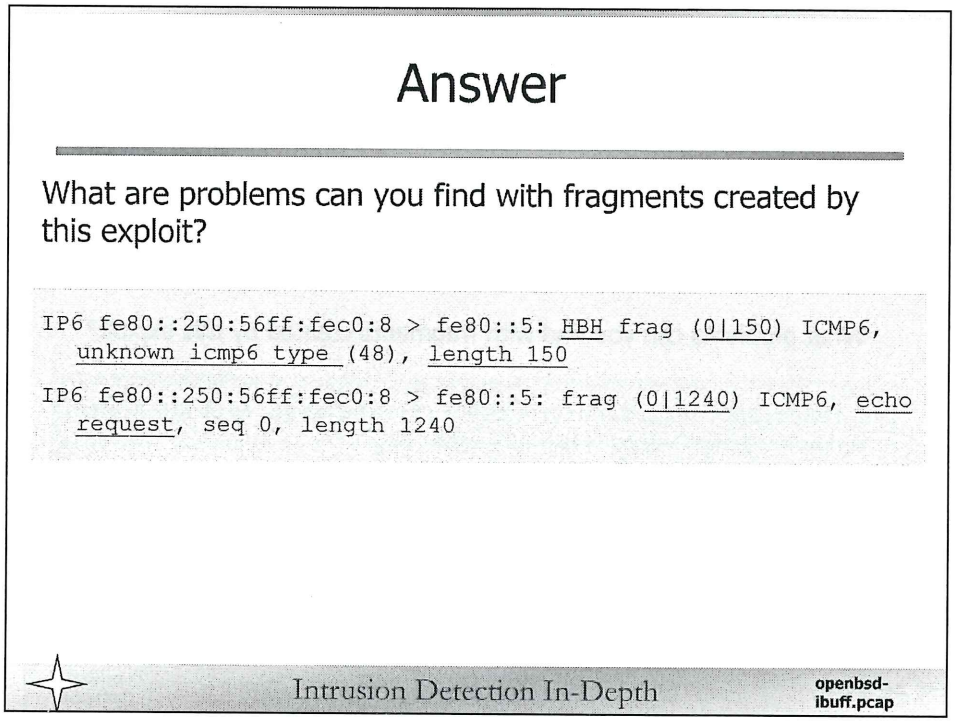


An exploit involving overlapping IPv6 fragments that targeted unpatched versions of OpenBSD 4.1 and prior would cause the OS to crash or allow a compromise via a buffer overflow upon receiving such packets. There are several issues with these two packets captured from this exploit. Apparently, error handling for this particular combination of malformed packets was faulty, permitting the denial of service or worse yet, later on, a buffer overflow.

What issues can you find simply by looking at the tcpdump output?

To see the output, enter in the command line:
wireshark openbsd-ibuff.pcap

You may find that the tcpdump output presents a more succinct view of the packets, permitting you to see the issues more easily than using Wireshark. For all of Wireshark's merits – brevity is not one.



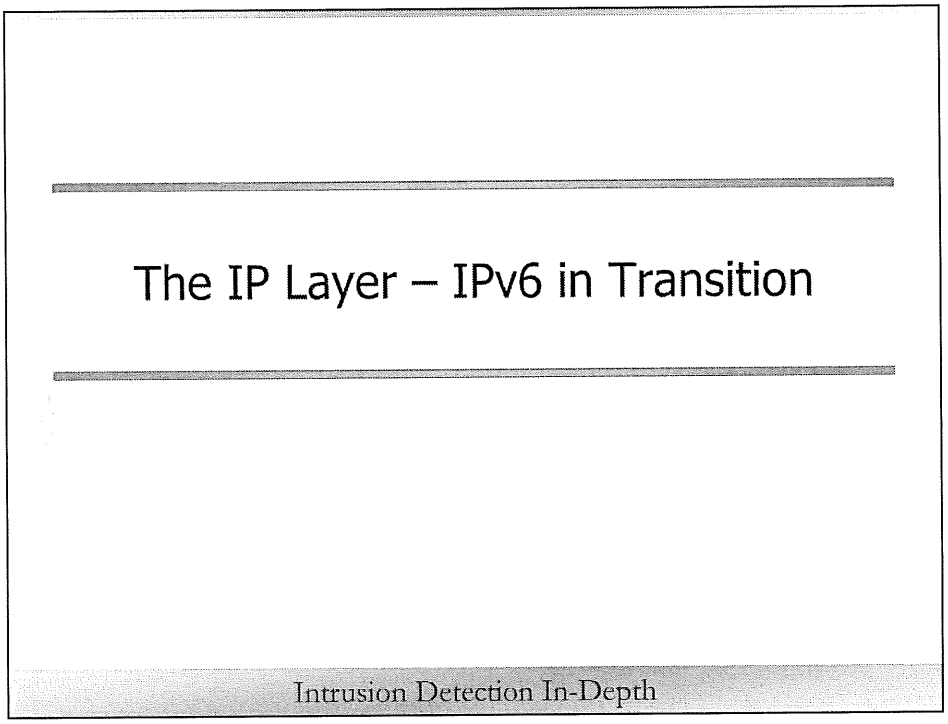
There are many "non-standard features" of the packets shown in tcpdump. The two hosts involved are source IPv6 address fe80::250:56ff:fec0:8 and the destination of fe80::5. The first packet is a fragment. The hop-by-hop header is the first extension followed by a fragment extension. Recall that the hop-by-hop is an unfragmentable extension header that should be found in every fragment. So far, all is okay. Next we find that the fragment has an offset of 0 and a length of 150. And, as we learned earlier, tcpdump does not give any indication whether or not the MF flag is set, though it is.

Do you see anything wrong with the length of the first fragment? Remember that the length of the fragment payload (whether data alone or the protocol header in the first fragment and data) should be a multiple of 8? 150 is not evenly divisible by 8. This necessarily means that the fragment that follows will either overlap or have a gap at its starting offset. Another problem is that ICMPv6 follows the fragment header, however there is an invalid ICMPv6 type in the ICMPv6 header.

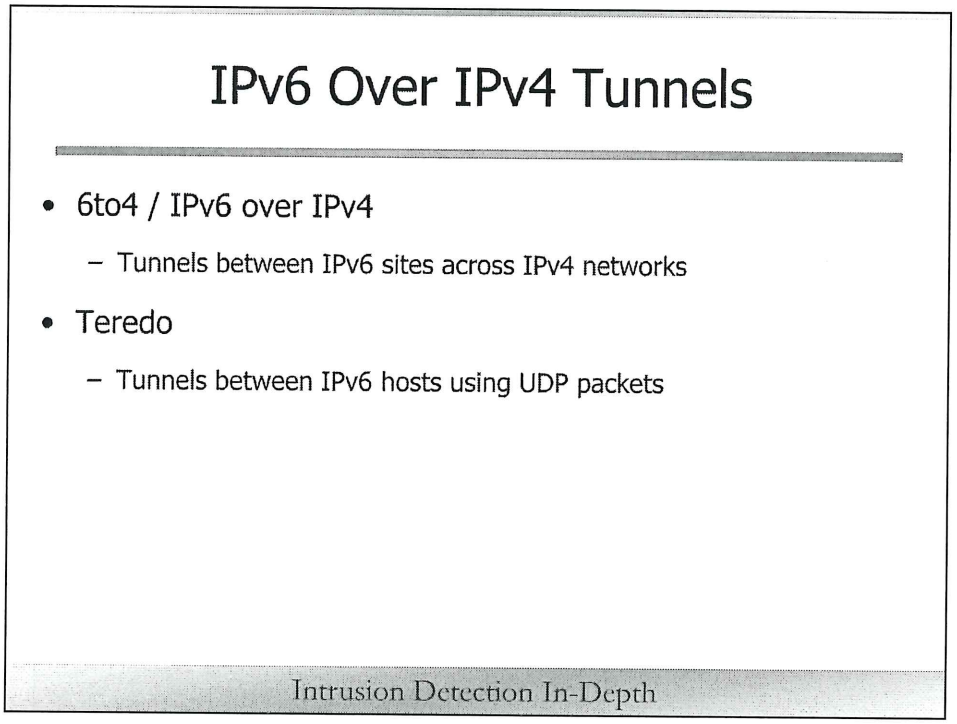
Moving on to the second fragment, we find more issues. The hop-by-hop extension header should follow the IPv6 header on the second fragment since as we learned, it is not fragmentable. However it is not present. The second fragment has an offset of 0 that overlaps the first fragment. We knew that either an overlap or gap would be present because the first fragment payload length is not evenly divisible by 8. And, as you learned, according to guidance, overlapping fragments should be discarded. An ICMPv6 echo request is associated with the second fragment, however only the first fragment should carry the embedded protocol header. The first had no valid ICMPv6 message type, yet the second does so they appear to have each other's ICMPv6 header/payload.

There is plenty wrong with these IPv6 packets and unless there is error code to properly handle all of these non-standard conditions, opportunities for denial of service and exploits present themselves. The pcap available for you to view for this output does not contain the shell code exploit. This payload caused a denial of service only.

To see the output, enter in the command line:
wireshark openbsd-ibuff.pcap



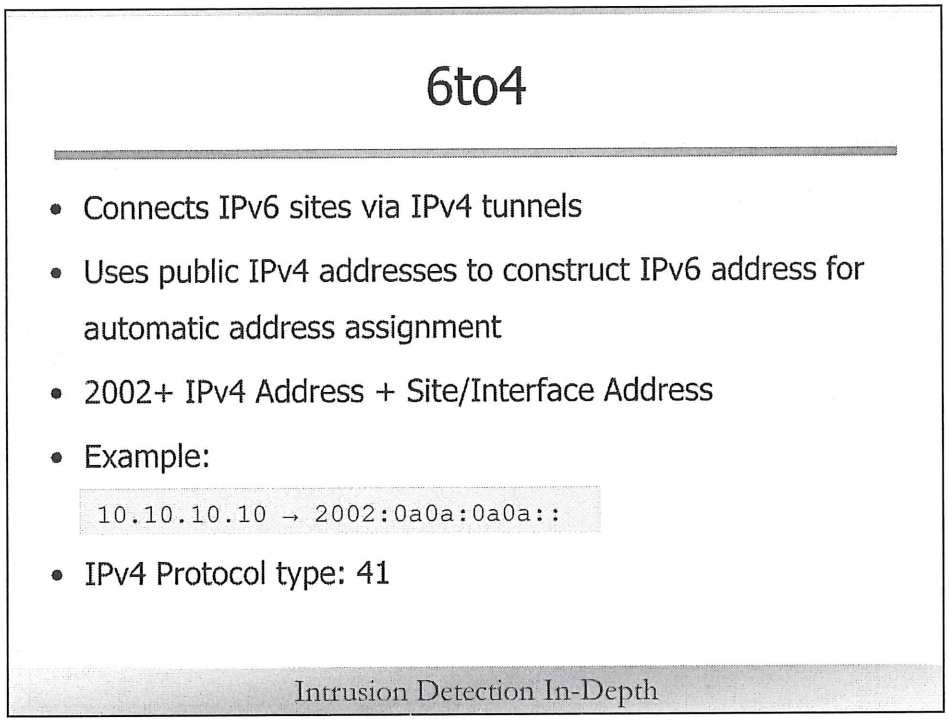
This page intentionally left blank.



IPv6 will have to cooperate with IPv4 for the foreseeable future. As individual networks implement IPv6, they will have to communicate via tunnels that cross the legacy IPv4 networks. Different technologies have emerged for this purpose.

6to4, or IPv6 over IPv4, uses a specific IPv4 protocol to tunnel IPv6. A simple IPv4 header is added or removed by respective gateways. This protocol requires both the sending and receiving networks to implement such a gateway.

Teredo is a protocol introduced by Microsoft. It allows IPv6-capable hosts to discover IPv6 hosts and to communicate with them via IPv4 Peer-to-Peer connections. As a UDP protocol, it relies on the encapsulated protocol for reliability.



6to4 can be used to link various IPv6 networks across IPv4 links. In order to route the packets across an IPv4 network, the IPv4 address of a host is used to derive the IPv6 address.

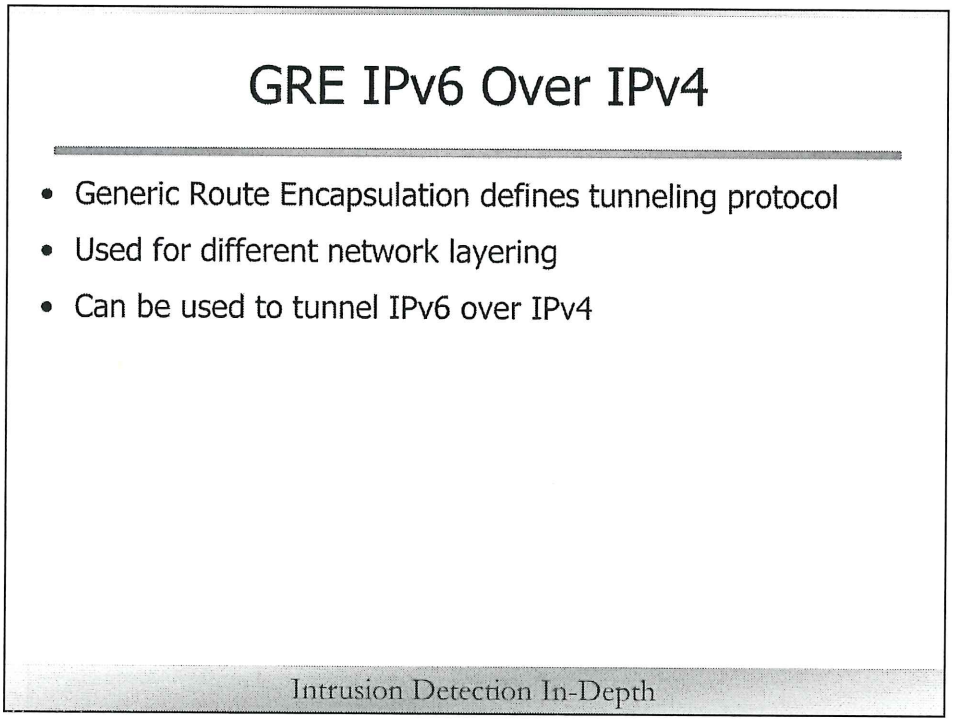
For example, if a host's IPv4 address is 10.10.10.10, then the IPv6 address of the host would use the prefix 2002:0a0a:0a0a::. A host trying to communicate with 2002:0a0a:0a0a:: will now encapsulate the IPv6 packet in an IPv4 packet addressed to 10.10.10.10.

In order to route traffic between IPv6 networks, only one IPv4 address is required for each network. Hosts in each network will use the same prefix (e.g. 2002:0a0a:0a0a).

Note that IP protocol 41 is used by 6to4 as well as by the 6over4 protocol. Both are very similar. However, 6over4 requires IPv4 multicast to work, while IPv6 multicast traffic will be transmitted as IPv4 multicast traffic. Multicast routing is not universally available, making 6over4 a less popular choice. For 6over4, the address prefix is "'E80::/64"' The IPv4 address is appended (e.g. FE80::0a0a:0a0a).

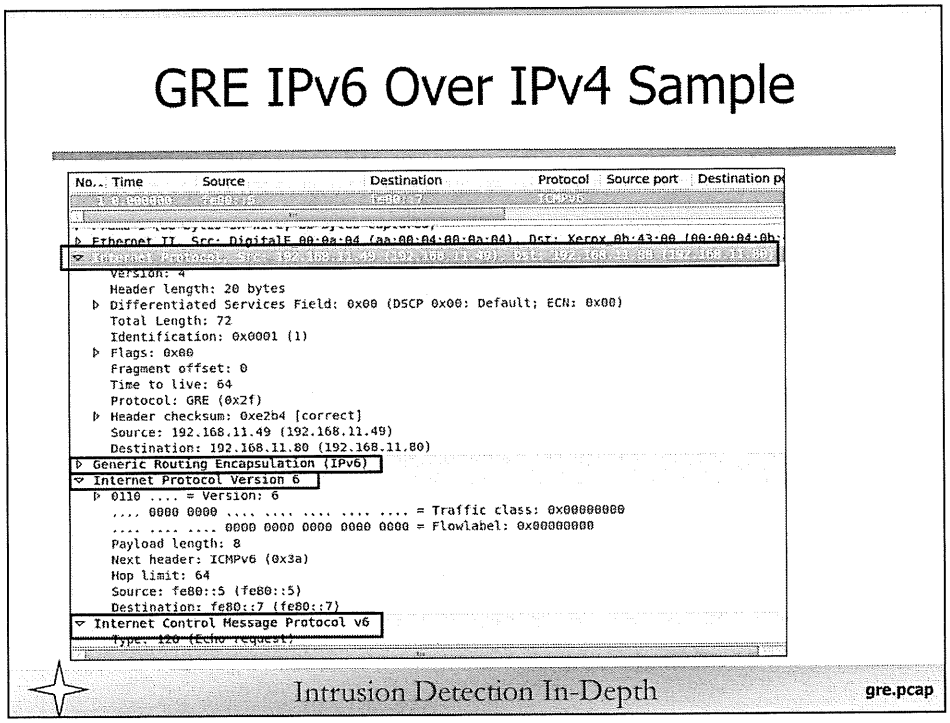
Check out the SANS Internet Storm Center 6/4 or 4/6 address conversion tool at:

http://isc.sans.edu/tools/ipv6.html



Another type of tunnel that can carry IPv6 packets over IPv4 is called Generic Routing Encapsulation. As the name implies, GRE tunnels can be used to transport many different protocols over IP.

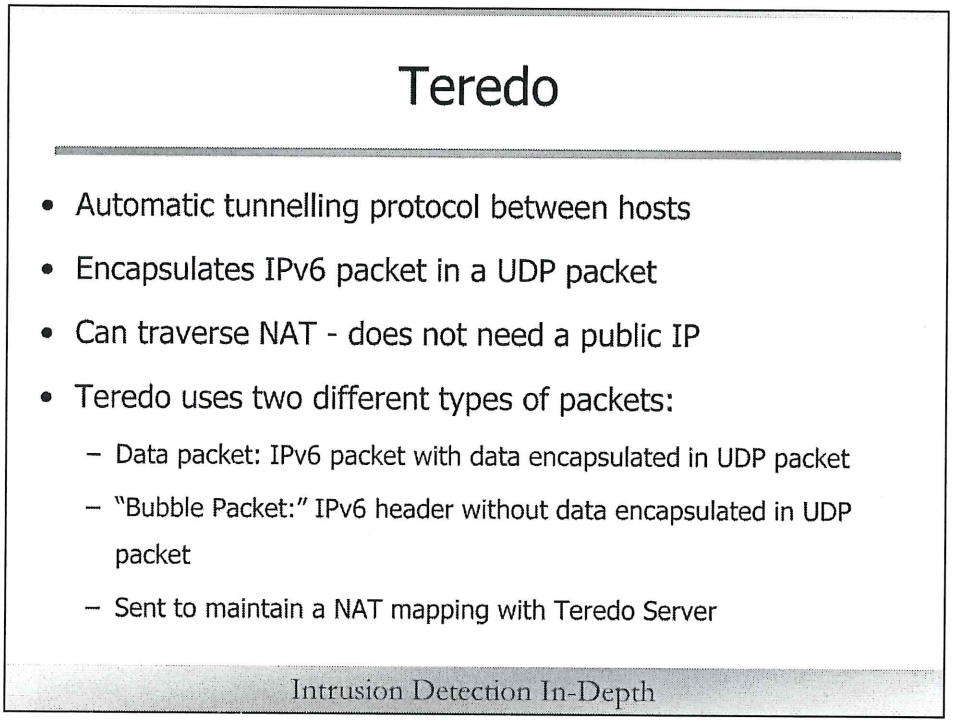
The tunnel software performs the encapsulation and de-encapsulation. Once proper network routing is set up, the IPv6 packets are routed to and from the tunnel and layered or stripped of the IPv4 header depending on the source and destination of the traffic.



The packet above shows an ICMPv6 echo request carried over IPv4. The outer layer is the IPv4 header with IPv4 addresses of 192.168.11.49 and 192.168.11.80 respectively for source and destination. These are IPv4 addresses associated with either end of the GRE tunnel. The next header value is 0x2f or decimal 47 that represents the protocol number associated with GRE. The GRE portion and the entire IPv6 packet follow. The IPv6 source is fe80::5 and the destination is fe80::7. These are the respective IPv6 addresses of hosts communicating on either side of the GRE tunnel.

To see the output, enter at the command line:

wireshark gre.pcap



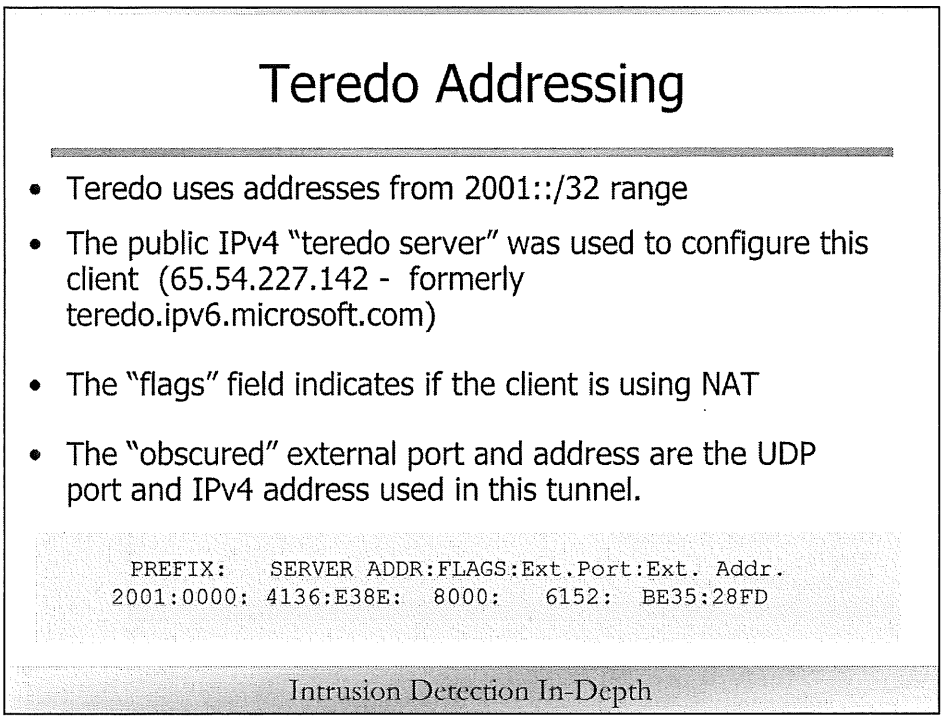
Teredo tunnels are established automatically, without any user intervention or configuration. Once the host discovers one Teredo-capable system, it will use it to discover more. Each host will maintain connections to a number of different Teredo hosts and these hosts will update each other about the state of new hosts.

In order to traverse firewalls and gateways easily, IPv6 packets are wrapped into UDP packets. While the IPv6 address of the host has to be unique, the IPv4 address does not have to be unique. Hosts with "unroutable" IP addresses behind NAT gateways will be able to participate.

Teredo hosts behind gateways will automatically send "Bubble" (keep-alive) packets to keep the connection across the gateway open.

Details:

http://technet.microsoft.com/en-us/library/bb457042.aspx

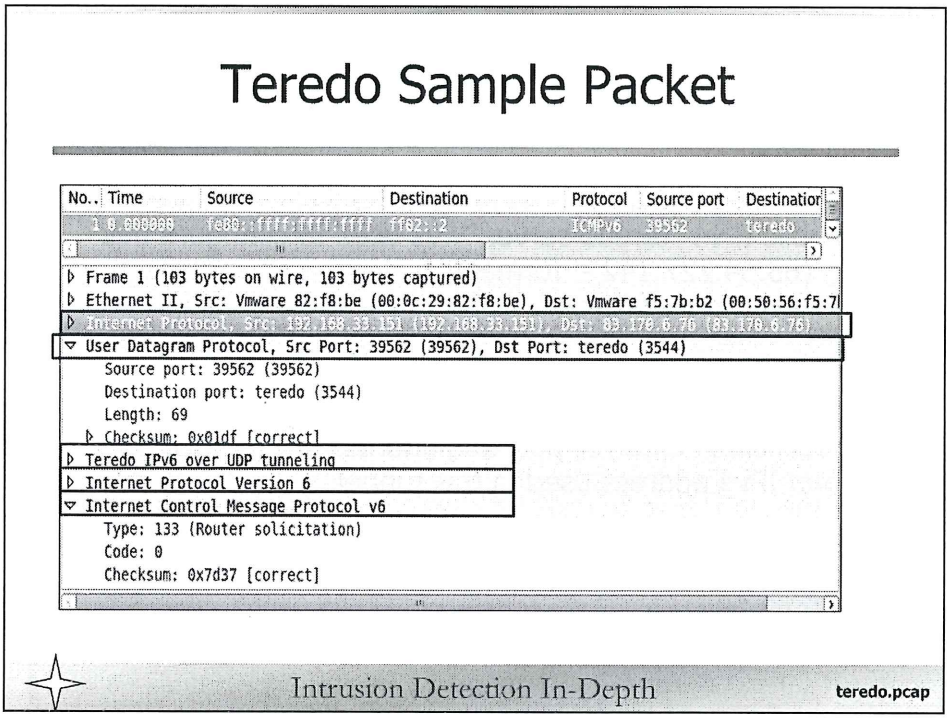


Teredo addresses always start with "2001". The original standard Teredo prefix was 3FFE:831F::/32 and was used by Windows XP and Windows Server 2003. However, RFC 4380 revised this to 2001::/32.

This is followed by the IPv4 address of the server used to configure this Teredo host. In this case 0x4136:E38E translates to 65.54.227.142 that resolved to teredo.ipv6.microsoft.com when this particular packet was captured . This was a public Teredo server that could be used to assist in Teredo configuration of a network where there was no Teredo server.

The Flags are used to indicate if a host is using NAT or not. The high order bit is set if the host is using NAT. Since the high-order bit found in 8 is set, it is using NAT. The final fields are the external UDP port and IPv4 address used in this tunnel. The fields contain the UDP port and IPv4 address of the NAT device that is sending this traffic. However, they are "obscured". They represent the actual port or IP address after an exclusive OR operation with 0xFFFF. For instance, the NAT device UDP port is 40621. 40621 XOR 0xFFFF = 0x6152. Similarly, the 0xBE35 28FD is the NAT device address of 65.202.215.2.

The external port and IP addresses are obscured to keep a NAT device from translating them within the payload of the packets that they are forwarding.



This is a Teredo packet. First, there is an IPv4 layer that has UDP as its protocol. The UDP port of 3544 is associated with Teredo. Wireshark interprets this as "Teredo IPv6 over UDP tunnelling". The UDP payload consists of a Teredo Authentication header (not shown) that is an indicator used to protect Router Solicitation, followed by an IPv6 header that carries an ICMPv6 Router Solicitation.

The Teredo server or relay that receives this is responsible for extracting the IPv6 packet and sending it over an IPv6-aware network to the destination host. When a response is returned as payload in the UDP datagram, the Teredo client must interpret the IPv6 payload.

A Unix implementation of Teredo is available as miredo and can be found at:

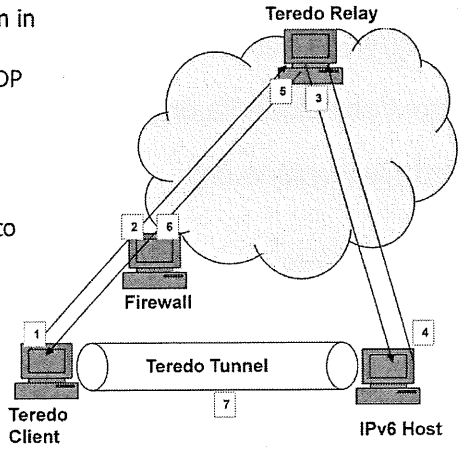
http://www.remlab.net/miredo

To see the output, enter in the command line;

wireshark teredo.pcap

Teredo Security Issue

1. Teredo Client sends IPv6 datagram in IPv4 UDP payload
2. Firewall permits outbound IPv4 UDP packet
3. Teredo relay unwraps IPv6 traffic from UDP payload, forwards over IPv6 network to IPv6 host
4. IPv6 host sends IPv6 traffic back to Teredo relay
5. Teredo relay encapsulates IPv6 in IPv4
   UDP payload and returns it over IPv4 network
6. Firewall allows the IPv4 UDP payload back in
7. Whole process = Teredo tunnel



Intrusion Detection In-Depth

Teredo comes installed on many later Windows versions. If the host attempts to make contact with a Teredo host – one with a 2001:: IPv6 address, Teredo becomes enabled. When this happens, the Teredo client becomes "qualified" by contacting a Teredo server over UDP port 3544 that gives the client a Teredo IP address to use to listen on a Teredo network interface. It then communicates with an IPv6 Teredo host using IPv4 UDP as the transport protocol and the Teredo client encapsulates an IPv6 packet within the UDP payload. If there is a stateful firewall at edge of the client's network, it will see the outbound UDP traffic and permit inbound return traffic. When a designated Teredo relay host receives the UDP datatgram, it de-encapsulates the IPv6 payload from the UDP data. It then forwards it over an IPv6 network to the destination. This process is reversed when the IPv6 host sends traffic back to the Teredo client.

This whole process essentially establishes a tunnel between the Teredo client and IPv6 host. This is fine when the Teredo client knowingly initiates a connection. However, if there is a malicious web server on the network that has a link to a Teredo host (2001:: address), the client unwittingly opens up a Teredo tunnel to the host referenced in the link. If this too is a malicious host, it now has a tunnel directly to the Teredo client bypassing the firewall. If the Teredo client host is not patched and has some kind of vulnerability, it is now exposed for the attacker to exploit.

More Information

- http://www.networksorcery.com
- http://en.wikipedia.org/wiki/IPv6
- http://www.iana.org

This page intentionally left blank.

IPv6 Tools

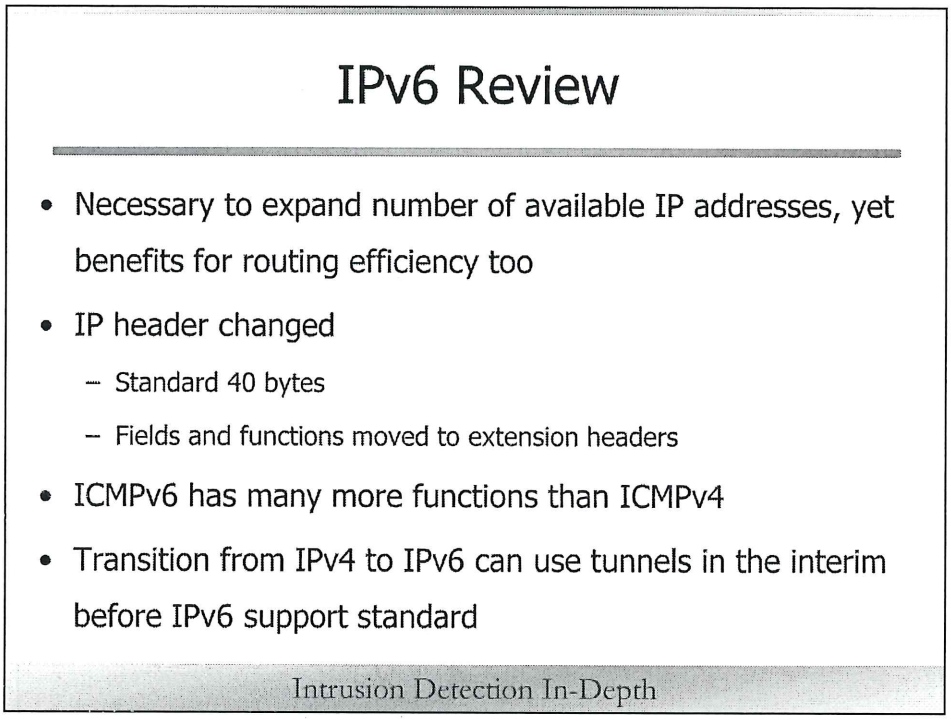
Most modern operating systems come with IPv6 support enabled and listening at some of the same ports used by IPv4 applications (http, ssh)

You may have to add an IPv6 address that is not a link local address to connect to the network

- nc6 – netcat 6
- ping6
- scapy
- ip6tables

Intrusion Detection In-Depth

This page intentionally left blank.



The proliferation of IP enabled devices has caused the exhaustion of IPv4 32-bit addresses. One of the reasons for IPv6 is the expansion of the available IP addresses, accomplished with a 128-bit address. Routing schemes have been simplified as well for more efficiency.

One of the most significant changes with respect to IPv4 is that the IPv6 header is limited to exactly 40 bytes. Extension headers are used in place of IP options fields and fragmentation has a separate extension header instead of being included in the standard header.

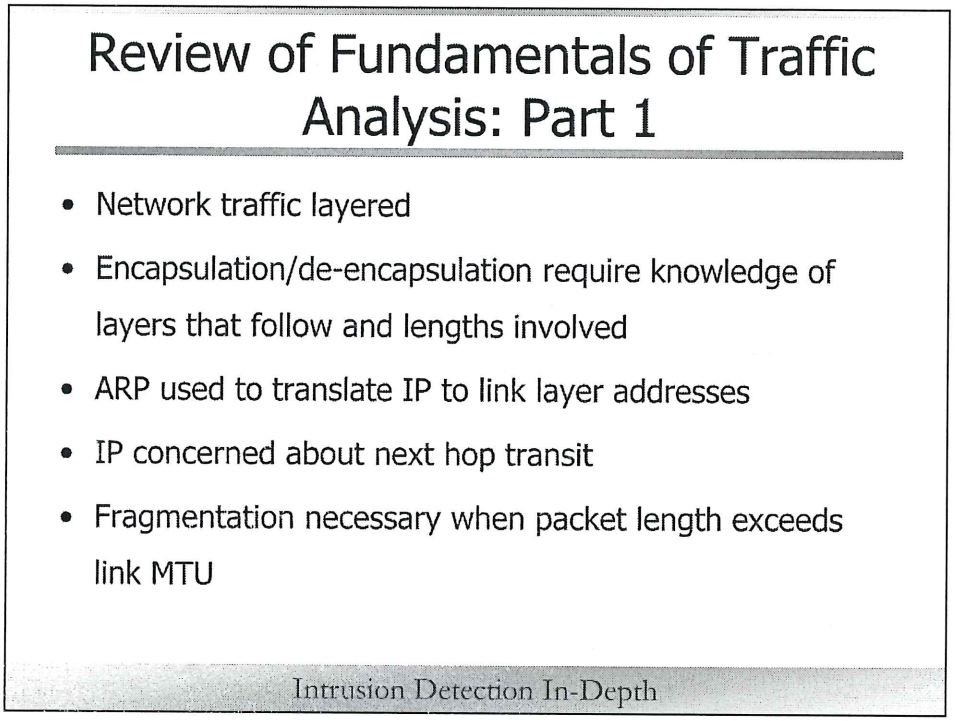
As the transition from IPv4 to IPv6 occurs, there are technologies to enable communications between the two. These include IP tunnels capable of carrying IPv6 packets over IPv4 packets where one or both gateways communicate using IPv4 only and endpoints are IPv6 enabled. Teredo provides IPv6 host communication by tunnelling IPv6 packets over IPv4 UDP.

IPv6 Exercises

Workbook

Intrusion Detection In-Depth

This page intentionally left blank.



Let's highlight some of the topics covered in Day 1. We examined the concept of layering of network data, including the processes of encapsulation and de-encapsulation. When decoding traffic, both software and humans must know how to perform interpretation using fields and values that contain current or following protocol identification. There also must be some designation of where protocols and fields begin and end. There may be an implicit unstated fixed length, a supplied variable length, or a computed length.

We examined the role of ARP to manage IPv4 to link layer addresses for local network communication. We learned that ARP is not secure and can be used to cause denial of service or MITM attacks. IPv6 has similar support and issues using Neighbor Discovery Protocol.

The next layer up on the TCP/IP model is IP. The function of IPv4 and IPv6 is to route a packet closer in hops to its ultimate destination. There are notable changes in IPv6, including a fixed length header and extension headers.

We discovered that fragmentation is employed when a packet is larger than the MTU of a link it needs to traverse. All fragments have an identifier to associate them with other fragments in the same fragment train, an offset to designate their place among the others, and finally a MF flag to inform whether or not other fragments follow.

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