Intrusion Detection using *tcpdump*

- ***tcpdump*** is a **network monitoring** application that can capture packet traffic on a broadcast network.

- The traffic of interest can be specified by command line arguments when tcpdump is run and the utility will only dump the required traffic.

- It supports a variety of output options but by default it outputs packet headers in a user readable format to the standard output.

- The size of data from a packet to be dumped is 68 octets by default. This can be altered by specifying the option **-s snaplen**, where snaplen is the length of a packet to be dumped.

- It records network traffic by the use of a host computer's **packet filter**. A packet filter is an operating system service for selectively recording network packets. This is referred to as packet **capture**.

- tcpdump is implemented using the Packet Capture library (**libpcap**), which recognizes a variety of packet capture systems on different operation system platforms.

- An example of a packet capture system is the **Berkeley Packet Filter** (BPF). The link-level device driver calls BPF when a packet arrives at the network interface before it makes a decision whether the packet is addressed to the local host and should be passed up to the system protocol stack.

- The BPF feeds the packet to the tcpdump process's filter that is specified by the user and is responsible for deciding whether a packet is to be accepted. If the packet is to be accepted, BPF copies the requested amount of data to the buffer associated with the filter and timestamps it.

- Timestamps measured by tcpdump are closer to packets' wire times than those acquired in the user-space by a process such as ping, since they are captured by BPF just above the link-layer device driver which directly communicates with the network interface card.

- Furthermore, since ping timestamps an ICMP request packet earlier than BPF and timestamps the ICMP reply packet later than BPF, the round-trip time reported by ping is always larger than that reported by the tcpdump running on the same machine.

Using tcpdump

- The ***tcpdump*** program is very useful tool that can be used to capture and read TCP/IP traffic on the network. Specifically it captures and prints out the headers of packets on a network interface.

- By default it captures and prints all the traffic the local subnet in a standard format. Command line options are provided for modifying the default behavior, either by collecting specified records, printing in a more verbose mode ***(-v)***, printing in hexadecimal ***(-x)*** or writing records as "raw packets" to a file ***(-w)*** instead of printing as standard output.

- We can also design **tcpdump filters** and specify packets to be captured. Rather than gather all traffic passing on the local subnet, tcpdump can be instructed to capture packets with very specific characteristics.

- Examples such filters would be to capture only TCP packets, or packets to a given port, say telnet, port 23. We can also limit the scope of packets captured to a specific IP address or hostname.

- Combinations of protocol details can be used to selectively capture packets. Just about any field in an IP datagram, including the actual data payload, can be used to select the packets that are captured.

- ***tcpdump*** has a default standard output based on the protocol (TCP, UDP, ICMP) of the record that is displayed.

- The following is a sample ***tcpdump*** udp output:

```
  Timestamp        source.port           dest.port     udp bytes
21:49:18.485000 ithaca.olympus.728 > valhalla.bcit.111: udp 56
```

timestamp: hour:minutes:seconds.fractions of seconds
source.port: source IP/hostname.source port
dest.port: destination IP/host.destination port
udp: may or may not expressly label the udp protocol
bytes: number of bytes of udp data (payload)

- The source information includes the **source host** name, **ithaca.olympus**, or IP number depending upon whether the IP can be resolved. To save dns query delays, ***tcpdump*** can be run with the ***-n*** paramter.

- The hostname is followed by a period and the **source port**, in this case **728**.

- Immediately following the greater than sign is the **destination host** or IP, **valhalla.bcit** followed by a period, and then the **destination port**, in this case port **111** (this happens to be the ***portmapper*** or ***sunrpc*** port).

- Next we see the word "**udp**", specifying the protocol. Note that not all udp records will be specifically labeled as such. DNS, or port 53, is an exception.

- The final field indicates the number of bytes found in the packet payload.

- The following is a sample ***tcpdump*** tcp output:

```
timestamp   source.port   dest.port flags  beginning:ending     bytes options seq#      seq#


21:49:18.485000 zeus.net.1173 > dns.net.21: S 62697789:62697789(0) win 512
```

flags: tcp flags (PSH, RST, SYN, FIN)
beginning seq #: for the initial connection, this is the initial sequence number (ISN) from the source IP
ending seq #: this is the beginning sequence number + data bytes
bytes: data bytes (payload) in the tcp packet
options: options that the source host advertises to the destination host

- The tcp output is identical to the udp record as far as timestamp, source and destination host and port.

- What distinguishes the tcp format from the others are the **tcp flags**, **sequence numbers**, **acknowledgements**, **acknowledgement numbers**, and **tcp options**.

- In this trace the flag **SYN** or **S** is set following the destination port of 21 (ftp). The SYN flag indicates a request to begin a tcp session.

- Other possible flag values are **P** for **PUSH** (sends remaining data), **R** for **RESET** (abort a connection) and **F** for **FIN** (gracefully terminates a connection).

- A period in the flag field simply indicates that none of the PUSH, RESET ,SYN or FIN flags are set.

- Next is the starting sequence number. Sequence numbers are used by tcp to achieve reliable packet delivery. In this case, since this is an initial connection, it is known as the initial Sequence Number or **ISN**.

- The ending sequence number is the sum of the initial sequence number plus the number of tcp data bytes sent in this tcp segment.

- A **SYN** connection sends **no data bytes**, as represented by the zero in parentheses. Data will not be sent until the two hosts complete the three-way handshake.

- Finally, there is a tcp options field. In this record, we see that **zeus.net** is advertising a **sliding window** size of **512 bytes**.

- A sliding window is used by one host to inform a remote host that it has maximum receive buffer size of 512 bytes.

- If **dns.net** is a more powerful, larger host, it will have to slow itself down and regulate the data it transmits so as to not overrun the buffer size of **zeus.net**.

- The next few slides provide examples that illustrate the basic details displayed by ***tcpdump***.

<u>Writing tcpdump Filters</u>

- One of the most powerful features of ***tcpdump*** is its ability to use filters to narrow down the type of packet captures to be displayed. Often, we will want to examine certain fields in the IP datagram for signs of malicious activity directed at our network.

- If we want wanted to capture specific types of packets, all we have to do is write a filter that uniquely specifies that type of packet. Then we execute ***tcpdump*** with the **filter option**, and tcpdump will use the filter as part of the packet selection criteria.

- Filters need to specify an item of interest, such as a **field** in the **IP datagram**, for record selection. Such items might be part of the IP header (the IP header length, for example), the TCP header (TCP flags, for example), the UDP header (the destination port, for example), or the ICMP message (message type, for example).

- Also provided are some macros for commonly used fields, such as "***port***" to indicate a source or destination port or "***host***" to indicate an IP numberor name of a source or destination host.

- Sometimes the fields we are interested in do not have macros, and so we must use the format of referencing a field by the protocol and displacement in terms of bytes into that protocol.

- ***tcpdump*** assigns a designated name for each type of header associated with a protocol. "***ip***" is used to denote a field in the IP header or data portion of the IP datagram, "***tcp***" for a field in the TCP header or data of the TCP segment, "***udp***" for a field in the UDP header or data of the UDP datagram, and "***icmp***" for a field in the ICMP header or data of the ICMP message**.**

- This gives us a reference a field in a given protocol by its displacement in bytes from the beginning of the protocol header.

- For instance, ***ip[0]*** indicates the first byte of the IP datagram, which happens to be part of the IP header (remember to count starting at 0).

- ***tcp[13]*** is **byte 14** into the TCP segment, which is also part of the TCP header, and ***icmp[0]*** is the first byte of the ICMP message, which is the ICMP message type.

- The format of the filters is pretty straightforward. The basic syntax for a filter is:

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

- The relational operators that ***tcpdump*** recognizes are: **=, <, >, <=, >=,!=,** as well as the logical operators **and (&), or (|), not (!).**

- A simple example would be a filter to capture all **telnet** traffic:

  # tcpdump –i eth0 –x 'tcp[2:2] = 23'

- This tells ***tcpdump*** to look at the **tcp header**. We skip bytes zero, and one (the source port), and look at bytes two and three (the destination port). If this two-byte field contains the value **23**, this is a telnet packet.

- To simplify things, ***tcpdump*** also includes some built in **macro's** that make writing filters even simpler. The following filter accomplishes the same thing as the filter above:

  # tcpdump –i eth0 –x 'tcp and dst port 23'

- The following filter can be used to capture NFS traffic:

  # tcpdump –i eth0 –x 'ip and udp port 2049'

- If we just want to capture **tcp** protocol packets:

  # tcpdump –i eth0 –x 'tcp'

- or

  # tcpdump –i eth0 –x 'ip[9:1] = 6'

- If we want to get more specific, and capture only tcp **SYN** packets, we can use the filter:

  # tcpdump –i eth0 –x 'tcp[13] & 2 != 0'

- This filter tells tcpdump to look at **byte 13** of the tcp header (remember to start counting at zero). Once you find byte 13, logically AND the value with 2 (0010), and compare the result of the AND to the value 1.

- If the result is zero, then the **SYN** flag is not set. If the value is 1, the SYN flag is set. Similarly, if I wanted to capture only packets that had the ACK flag set, I could use the filter:

  # tcpdump –i eth0 –x 'tcp[13] & 0x10 != 0'

- The main reason why we want to isolate this field is to test for the presence of IP options. The normal IP header is 20 bytes, or five 32-bit words.

- That means that an IP header that might contain a dangerous IP option, such as source routing, would have a length of greater than 5 found in this field. IP options are almost never used any more for anything other than evil intent, so we want to know whether IP options exist.

- Here is another example that is a complete filter to capture a signature of an IP datagram that has IP options set:

  ```
  # tcpdump -i eth0 -x 'ip[0] & 0x0f > 5'
  ```

- It is possible to string several filters together using parenthesis and logical operators to create very powerful and effective filters.

- Consider the following filter (annotations are shown after the # (comment) sign):

```
# Capture any packets that are
ip and                              # IP packets, AND
(
(ip[12:4] = ip[16:4])               # the source IP == the destination IP
or                                  # OR
    ((not src net 192.168) and      # if the source net is not 192.168 AND
    (
        (ip[19] = 0xff) or          # the destination is all 1's broadcast OR
        (ip[19] = 0x00) or          # the destination is all 0's broadcast OR
        ((ip[6:1] & 0x20 != 0 ) and # if MF is set, AND
        (ip[6:2] & 0x1fff = 0)) or  # this is the first fragment OR
          (net 0 or net 127 or net 1) or  # if the destination net is 0, 127, or 1
                                    # OR
        (ip[12] > 239) or           # if the destination IP is class D or E OR
        ( ((ip[0:1] & 0xf) > 5) )   # if the packet contains IP options
    ))                              # (more than five 32 bit words in the IP
)                                   # header == options in use)
```

- Typing in complex filters in on the command line is a tedious task at best. Therefore we would put such a filter in a file (for example ***ipfilter***), and call tcpdump with the –***F*** filter option:

  ```
  # tcpdump –i eth0 –x –F ipfilter
  ```

- It is important to note that the filter files **must not** contain comments. The comments above are there just for the purposes of explanation and must be removed before invoking ***tcpdump*** with the file.

- Some of the telltale indications in the IP header that you might be a target of reconnaissance include traffic sent to your **broadcast address**, **fragmentation**, and the **presence of IP options.**

- You should never see legitimate traffic directed to your broadcast address from outside your network, and that should be blocked at the external firewall.

- Fragmentation is a natural enough byproduct of a datagram destined to your network via a network that permits a larger MTU than one of the routes it took to get to your network.

- But, fragmentation can also be used for denial-of-service attacks or to try to bypass notice by an IDS or routers that cannot keep track of state.

Detecting Traffic to the Broadcast Addresses

- A broadcast address is defined as one with a final octet of 255 or 0. This includes most broadcast addresses subdivided on classic byte boundaries. The destination address is found in bytes 16 through 19 (32 bits) of the IP header.

- Of concern to us is the final octet, or **byte 19.** We can describe the broadcast addresses as follows:

    ip[19] = 0xff

    ip[19] = 0x00

- Or as a combined filter as follows:

  ip[19] = 0xff or ip[19] = 0x00

- As a matter of course hexadecimal notation is used. But, the above can also be expressed in decinal as follows:

  ip[19] = 255 or ip[19] = 0

- Depending on where the host running the ***tcpdump*** filter is located, you might pick up broadcast traffic inside your network.

- Assume that the inside network is 192.168.x.x. To further refine this filter to examine only traffic directed toward your network from a foreign source, the filter as follows:

  not src net 192.168 and (ip[19] = 0xff or ip[19] = 0x00)

- The macro operator ***not***, is used to negate; and ***src***, to indicate the traffic originated from this source; and ***net***, to indicate a subnet.

- The above filter will capture any traffic that originates from a source network other than your own that is destined for the broadcast addresses.

Detecting Fragmentation

- All fragments in a fragmented packet sequence except the last one have the more fragments (**MF**) bit set. If we locate this field and test the MF bit to see if it is set, we can find most of the fragmented traffic directed to our network.

- Specifically, if you count into the IP header, you will find it in the **sixth byte**. It is the **third bit** from the left of the high order-bit.

- The mask needs to be 0010 0000, which is a hexadecimal 0x20. Thus our filter becomes ip[6] & 0x20 != 0.

- Another filter to detect the MF bit would match the fragmented datagrams but will miss the the last fragment (which has the 2nd bit set to 0):

  # tcpdump -i eth1 'ip[6] = 32'

- The last fragment has the first 3 bits set to 0, but contains values in the fragment offset field.

- The following filter will match all the fragments, including the last fragment:

  # tcpdump -i eth1 '((ip[6:2] > 0) and (not ip[6] = 64))'

  "ip[6:2] > 0" - returns anything with a value of at least 1
  "not ip[6] = 64" - omit datagrams with the DF bit set

- Note that because fragmentation is not always malicious, you are likely to generate false positives with this filter.

UDP Filters

- Many backdoors and Trojans use UDP ports. UDP is a very simple header to deal with as far as filter design is concerned. To detect UDP traffic, we select UDP ports to be monitored for suspicious activity.

- For example, to scan for traffic to port 31337, the filter is as follows:

  udp and dst port 31337

- The main effort is not so much designing the filter but in deciding which ports you want include, adding them to the filter, and keeping the filter current with the real world of ever-expanding UDP exploits.

- Consider a popular UDP application, traceroute. This tool maps a path to a destination host by attempting to send UDP datagrams to high-numbered ports of the destination host.

- If a host on your network is that destination host, you want to be alerted of the attempted or successful traceroute. Start by looking at UDP activity to ports in the 33000-33999 range for most of the traceroute activity.

- Note that the Windows version of traceroute use ICMP echo requests and replies, so this signature does not detect that activity.

- Also note that some versions of ***traceroute*** enable the user to provide command-line options, one of which is a destination port. Therefore, this filter might not capture all traceroute activity, but it will find most of the conventional activity.

- The UDP destination port number is found in bytes 2 and 3 of the UDP header. Question: "why don't we use the port macro rather than byte displacements?" For instance, why can't we use this filter:

    dst port >= 33000 and dst port < 34000

- The problem is that when ***tcpdump*** uses a range such as this and not one exact value, you have to express that field in terms of the primitive protocol and displacement and forgo the use of macros.

- The correct syntax to discover ***traceroute*** then becomes this:

  udp[2:2] >= 33000 and udp[2:2] < 34000

- The length option [2:2] will span bytes. We need to examine 2 consecutive bytes starting at byte 2.

- We can further limit the amount of traffic that this filter extracts by examining the TTL value along with the destination port.

- ***traceroute*** operates by manipulating the **TTL** value found in the IP header. It records the routers that it traverses and does so using an incrementing TTL value. Frequently you will see a **TTL of 1** on the sensor host running ***tcpdump*** before it crosses a router that will expire it. This is a classic traceroute signature.

- Therefore, we can further refine the traceroute filter to include the TTL value to eliminate some of the noise associated with discovering traceroutes.

- The **TTL field** is found in the IP header; it has no macro to reference it, but it is located in the **eighth byte**. The new filter is as follows:

  udp[2:2] >= 33000 and udp[2:2] < 34000 and ip[8] = 1

TCP Filters

- Filters for TCP traffic are mostly concerned with initial **SYN connections** and other types of anomalous **flag combinations** that might indicate some kind of **reconnaissance** or **mapping efforts**.

- We want to look for initial SYN connections because they inform us of attempted connections to a TCP port. This doesn't necessarily mean that they were successful.

- If the sensor is located outside a packet-filtering device that blocks access to the TCP destination port, it will never reach the host. We can glean a lot of intelligence by detecting this activity, the least of which is discovering rogue TCP ports that hosts on your network might be offering.

Filters for Examining TCP Flags

- The TCP flag bits are located 13 bytes into the TCP header. Because we are looking for individual bits in the bytes, some bit-masking must be performed to select the flag or flags to be examined.

- We start by writing a filter to extract records with the SYN flag alone set:

  tcp[13] & 0xff = 2

- The mask consists of all 1's. Why not use a mask of 0's in all fields except the SYN flag, i.e., (***tcp[13] & 0x02 = 2***)?

- By masking a bit with a 0, the resulting (byte) value is necessarily 0. The value bit could be 1, however, and the 0 mask would discard it.

- Suppose that you want to look at TCP segments with the SYN flag alone set. Now suppose that you have a TCP flag byte with both the **SYN** and **ACK** flags set. The binary value that you would see for the TCP flag byte would be **0001 0010**.

- If that were masked with **0000 0010,** you would end up with a result of **0000 0010**, which is **2.** This is a misleading result because the intent here is to capture those packets with **only** the **SYN bit set**.

- Therefore, masking with 0's in fields other than the SYN flag selects TCP segments with other flags set along with the SYN flag.

- To prevent this from occurring, you use the original filter and look for an exact value. This filter does not select records with other flags set along with the SYN flag.

- The following are some other TCP flag combinations that might be very useful to be aware of:

  tcp[13] = 0

- This indicates null scans with no flags set. This condition should never occur.

  tcp[13] = 3

- This indicates activity where both the SYN and FIN flags are set simultaneously.
- This is definitely an anomalous condition. Alter the filter to tcp[13] & 0x03 = 3, to detect any activity with both the SYN and FIN flags set, as well as any other flags set.
- In this case, you don't necessarily want to limit this to SYN and FIN alone.

  tcp[13] = 0x10 and tcp[8:4] = 0

- Indicates activity with the ACK flag set, but with an acknowledgement value of 0.
- Any TCP segment with the ACK flag on should have a minimum acknowledgement value of 1 that occurs during the three-way handshake.
- At least 1 sequence number has to be consumed to elicit a valid acknowledgement; otherwise no acknowledgement would be returned.
- This filter captures ***nmap*** operating system fingerprinting scans that sends TCP traffic to various destination ports with the ACK flag alone set, but a 0 value in the acknowledgement field.

  tcp[13] >= 64

- The two high-order bits in the TCP 8-bit flag byte are labeled reserved bits. These 2 bits should be 0's; if they are not, something is amiss.
- The first reserved bit is found in the 2, (64) position, and the second is found in the 27 (128) position.
- If either or both bits are set, the value for the TCP flag byte is greater than or equal to 64. ***nmap*** sometimes sets one or both of these bits to perform operating system fingerprinting.
- Most hosts reset these values to 0's, but some leave the set value. This is used by ***nmap*** to help classify the operating system behavior.

__**Capturing HTTP Traffic:**__

- Let us say we want to design a filter that will capture any packets containing HTTP GET requests.

- The HTTP request will contains a starting string similar to (which is 16 bytes counting the carriage return but not the backslashes)

  GET / HTTP/1.1\r\n

- Assuming no IP options are set, the GET command will use the bytes 20, 21 and 22. Options usually will take up to 12 bytes (12th byte indicates the header length, which should indicate 32 bytes).

- Therefore, in our filter we match bytes 32, 33 and 34 (remember to count from 0).

- The string "GET " in hex is: 47455420. Thus our filter becomes:

```
# tcpdump -i eth1 'tcp[32:4] = 0x47455420'
```

Monitoring Key Hosts and Servers

- DNS, Web, and mail servers tend to draw a lot of activity from attackers.

For example, if a network loses control of its DNS, it is wide open to an attacker so it is worth the time to give connection attempts to these systems some extra attention.

- Consider the following filter for a web server:

```
(dst host 192.168.1.1 and
(
   (tcp and ((tcp[13] & 2 != 0) and (tcp[13] & 0x10 = 0))
              and (not dst port 80))
    or
   (udp and not dst port 53 and not dst port 137)
    or
   (icmp and (icmp[0] != 8) and (icmp[0] != 0)
        and (icmp[0] != 3) and (icmp[0] != 11))
    or
   (not (tcp or udp or icmp))
 ))
```

- In the above filter 192.168.1.1 is the web server and it should only receive traffic to tcp port 80 (SYN only).

    - The following traffic is being ignored:
    - udp with dst port 53 or 137
    - icmp echo requests (8), echo replies (0),

destination unreachable (3), and time exceeded (11) error messages

- The idea is to cut down on the “noise” so we can focus only on the main traffic of interest.