CSC 432-Z1

Lab – Week 5

Professor: Ronny Bull

April 10, 2022

Joseph (JT) Cavallaro

Abstract

Scapy is a library for Python that allows programmers to create, send, sniff, and dissect network packets. It is able to create tools with the power to scan, analyze, probe, and/or attack networks.

"In other words, Scapy is a powerful interactive packet manipulation program. It is able to forge or decode packets of a wide number of protocols, send them on the wire, capture them, match requests and replies, and much more. Scapy can easily handle most classical tasks like scanning, tracerouting, probing, unit tests, attacks, or network discovery" (Scapy, 2022).

We will use the Interactive Tutorial in the docs, first within the Ubuntu VM terminal, then within the CORE network topology emulation we will continue the tutorial and review the results for both. Then we will create two Python scripts using commands from the tutorial and again review the results.

Introduction

In this lab, we will be using this Python library, Scapy, to customize the terminal, build a packet, stack the layers, read PCAP files, create a graphical dump, generate sets of packets, send packets, fuzz packets, inject bytes, send and receive packets, preform syn scans, and preform a tcp traceroute. To do this, we will run through the Scapy interactive tutorial on the docs and review the results. Then we will create a network topology using CORE with three nodes (PCs) and a Hub. Once this is created and configured, we will continue the tutorial from the Scapy docs, and review the results. Once we are done, we will create two Python scripts using commands from the Scapy tutorial. We will run these within the topology and use TCPDump to capture the data so we can review the results.

Python Module Scapy, Scapy, & Dependencies Install

In this section, we will be installing the Python module Scapy, a library that allows programmers to create, send, sniff, and dissect network packets. Along with this, we will also be installing Scapy, the source code itself, then installing all of the dependencies needed for Scapy to work properly.

Python Scapy

First, we need to install the Python Scapy module. Scapy comes in three different bundles, only Scapy, Scapy Basic, and Scapy Complete. Scapy Basic installs Scapy and IPython while the complete bundle installs Scapy and all of the main dependencies. To install Python Scapy, I used the command "sudo pip install --pre scapy[complete]" which was already installed on my VM (Figure 1).

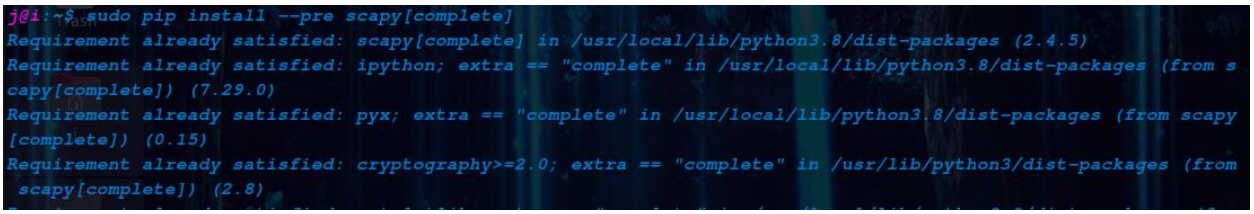

Figure 1 - Python Scapy Library Install

Scapy

Now that Python Scapy is installed, we can install the Scapy source code. We can access this at the Github page and the command “sudo git clone https://github.com/secdev/scapy.git” (Figure 2). Then, after changing directories to the newly created scapy directory and built and installed Scapy (Figure 2 & 3). To do this, I used the commands “cd scapy” and “sudo python3 setup.py build” and “sudo python3 setup.py build install” (Figure 2).

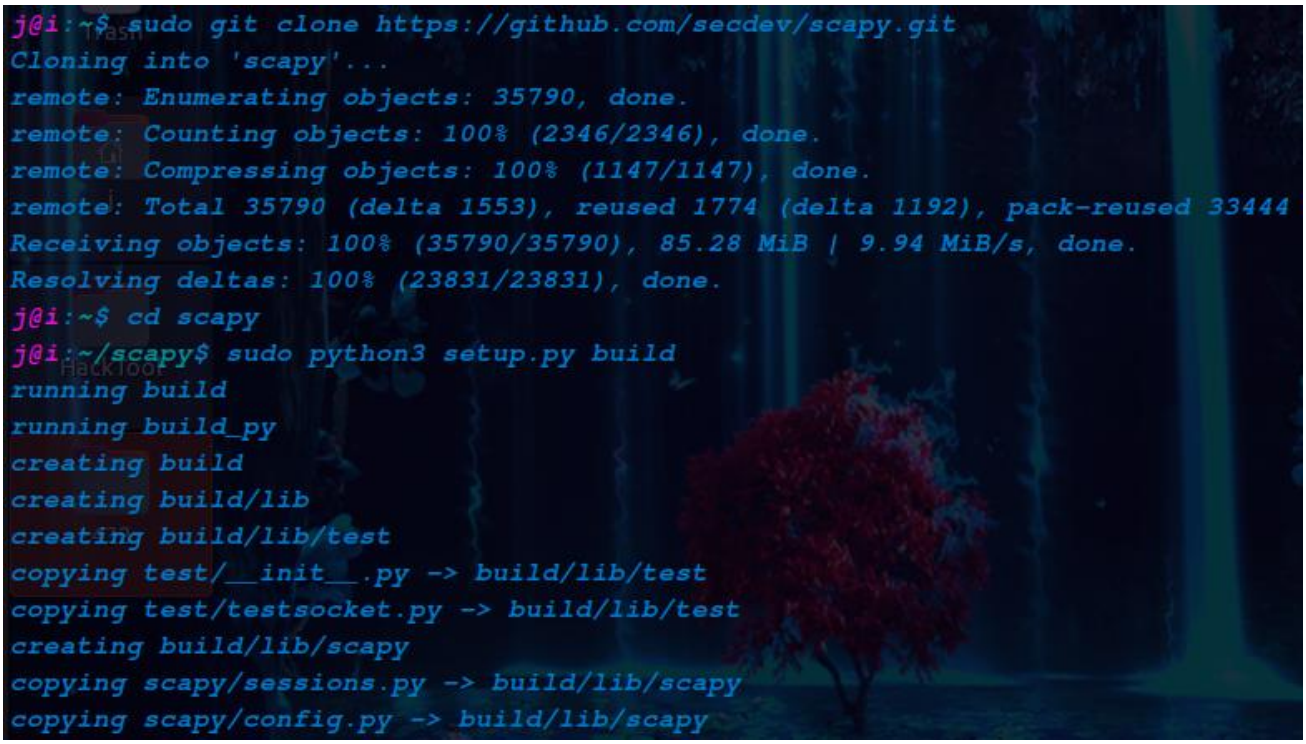

Figure 2 - Scapy Install - Steps 1 - 3

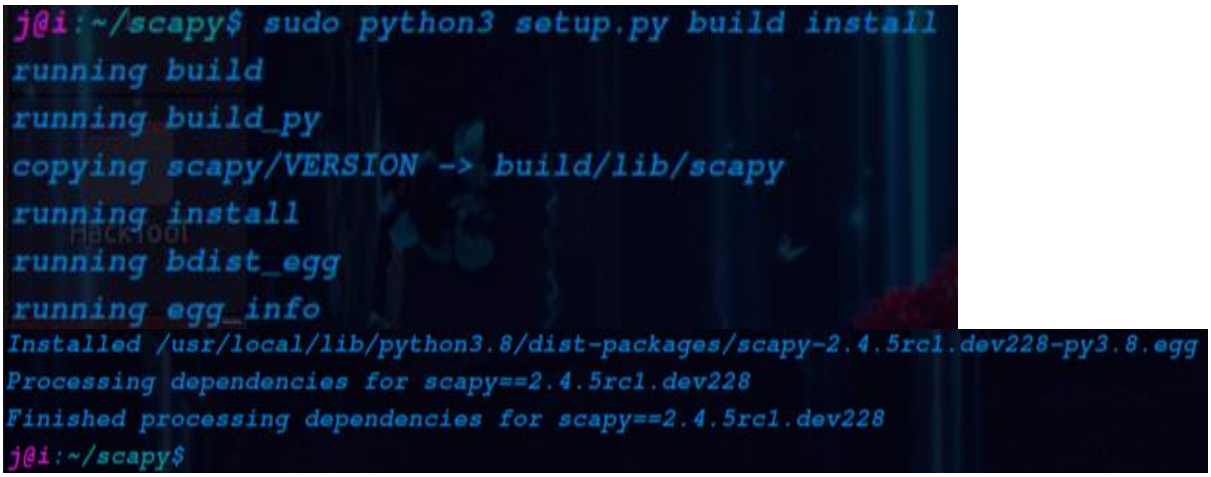

Figure 3 - Scapy Install - Step 4

Matplotlib

Now that the Scapy source code is built and installed, we can install the needed dependencies for Scapy to run properly. The first is Matplotlib, installed using the command "sudo pip install matplotlib" (Figure 4). This was already installed on my Ubuntu VM.

Figure 4 - Matplotlib Install

PyX, Graphviz & Python Graphviz

Now that Matplotlib is installed, we can install PyX, Graphviz and Python Graphviz. First is PyX, which was installed by using the command “sudo pip install PyX” (Figure 5). Then I installed Graphviz by using the command “sudo apt-get install -y graphviz” (Figure 5). Finally, I installed the Python Graphviz module by using the command “sudo apt-get install python3-graphviz” (Figure 6).

Figure 5 - PyX & Graphviz Install

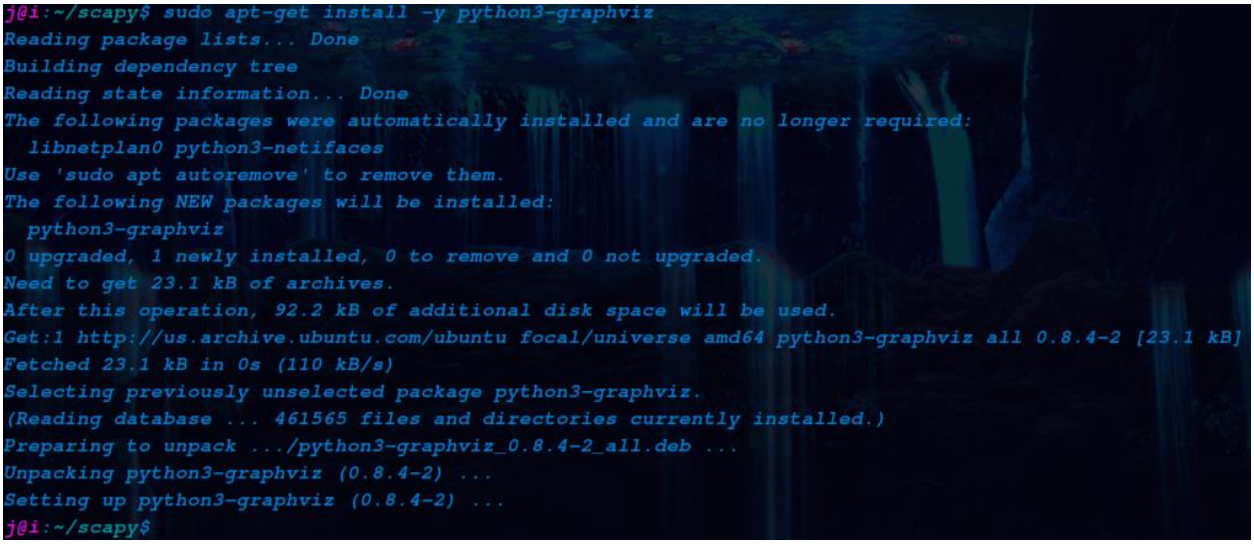

Figure 6 - Python Graphviz Install

ImageMagick

Moving on, we can now install ImageMagick. The first step is to install the Ubuntu package version of ImageMagick. To do this, I used the “sudo apt install imagemagick” which was already installed (Figure 7).

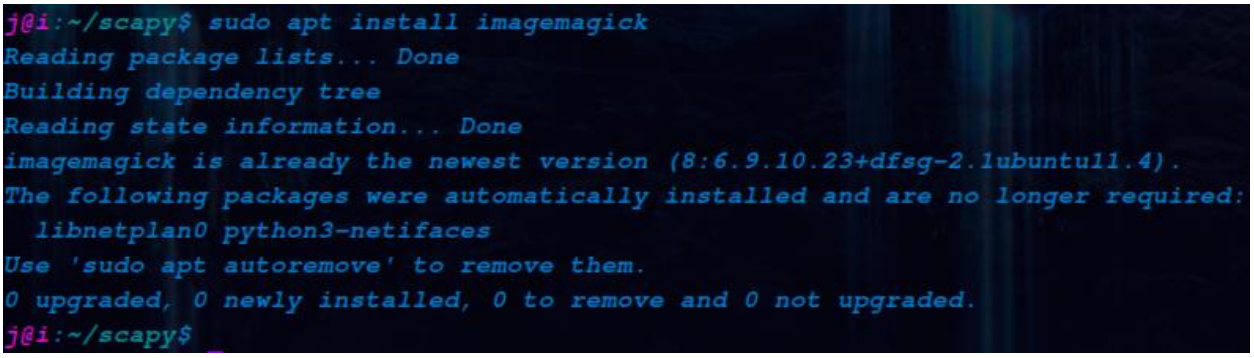

Figure 7 - ImageMagick Install - Step 1

Now that the Ubuntu package of ImageMagick, we can install the source code of ImageMagick by using the command “sudo wget https://download.imagemagick.org/ImageMagick/download/ImageMagick.tar.gz” (Figure 8).

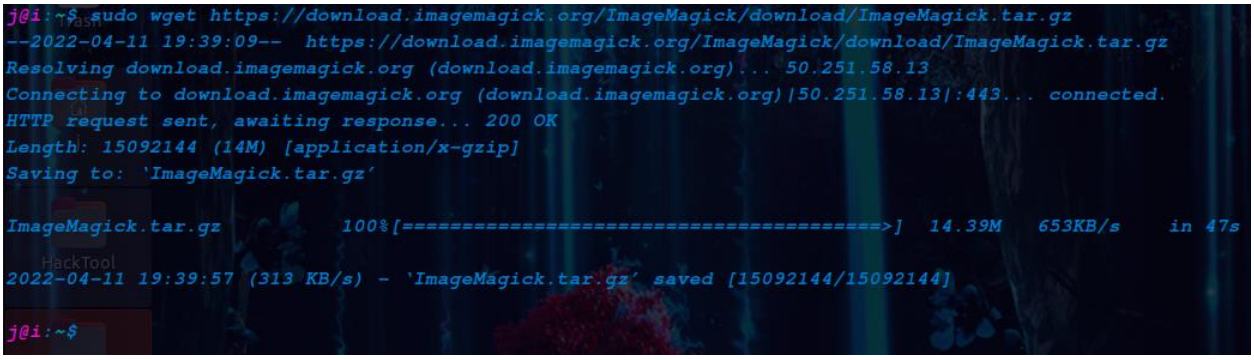

Figure 8 - ImageMagick Install - Step 2

Now that we download the tar.gz file, we can use the ‘tar’ command to extract it. Then change directories to the newly created ImageMagick folder and configured the source code. After the configuration completed, I needed to make and install the ImageMagick source code. This allows us to set ImageMagick to be used within the home directory. Then, to verify that ImageMagick is installed, we can use the identify command from withing the ImageMagick directory as well from the home directory (Figure 10). Below are the command, listed in order, used to complete the steps above:

- “sudo tar -axvf ImageMagick.tar.gz”
- “cd ImageMagick-7.1.0-29/”
- “sudo ./configure”
- “sudo make install”
- “sudo ldconfig /usr/local/lib”
- “identify”
- “cd”
- “identify”

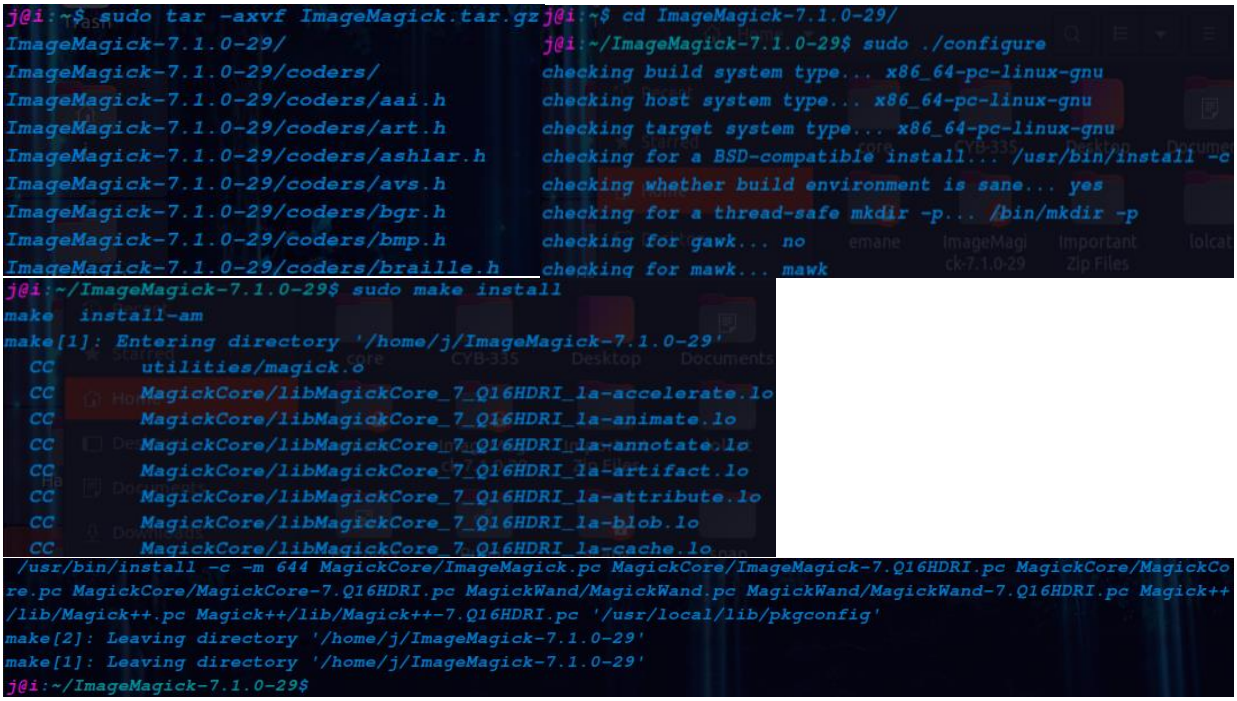

Figure 9 - ImageMagick Install - Steps 3 - 6

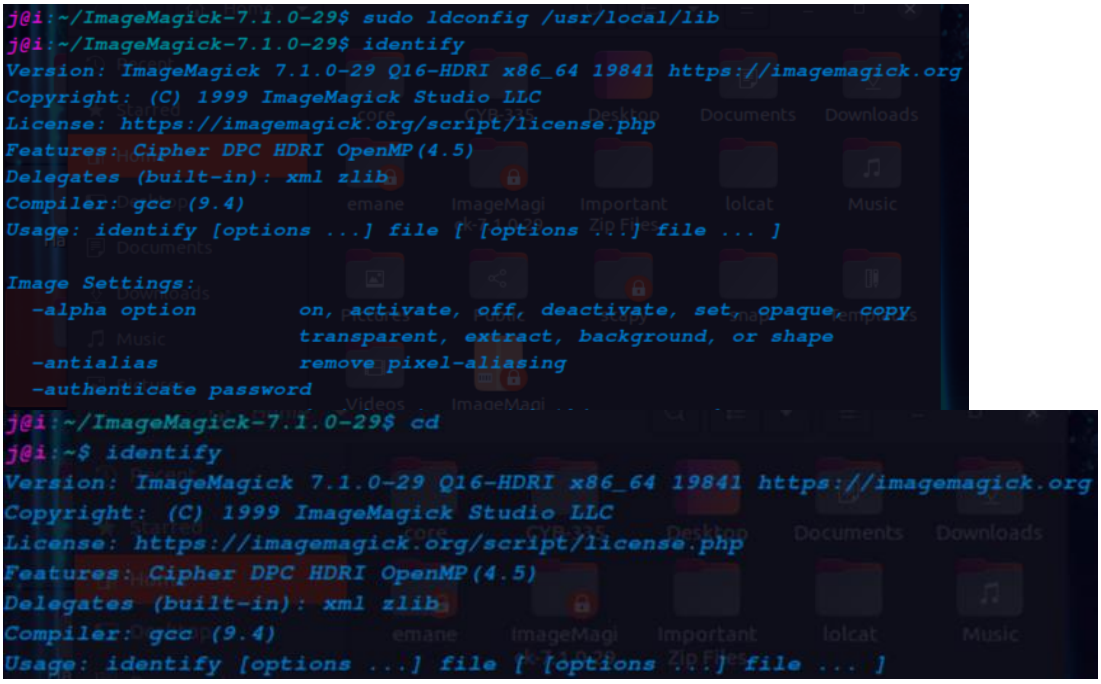

Figure 10 - ImageMagick Install - Steps 7 - 10

vPython, pyzmq, libzmq3-dev & Python-jinja2

Now that ImageMagick is installed, we can install vPython, pyzmq, libzmq3-dev and Python-jinja2. First we will install vPython by using the command “pip install vpython” (Figure 11). I noticed that pyzmq and jinja2 were out of date (red text in Figure 11), so I updated them.

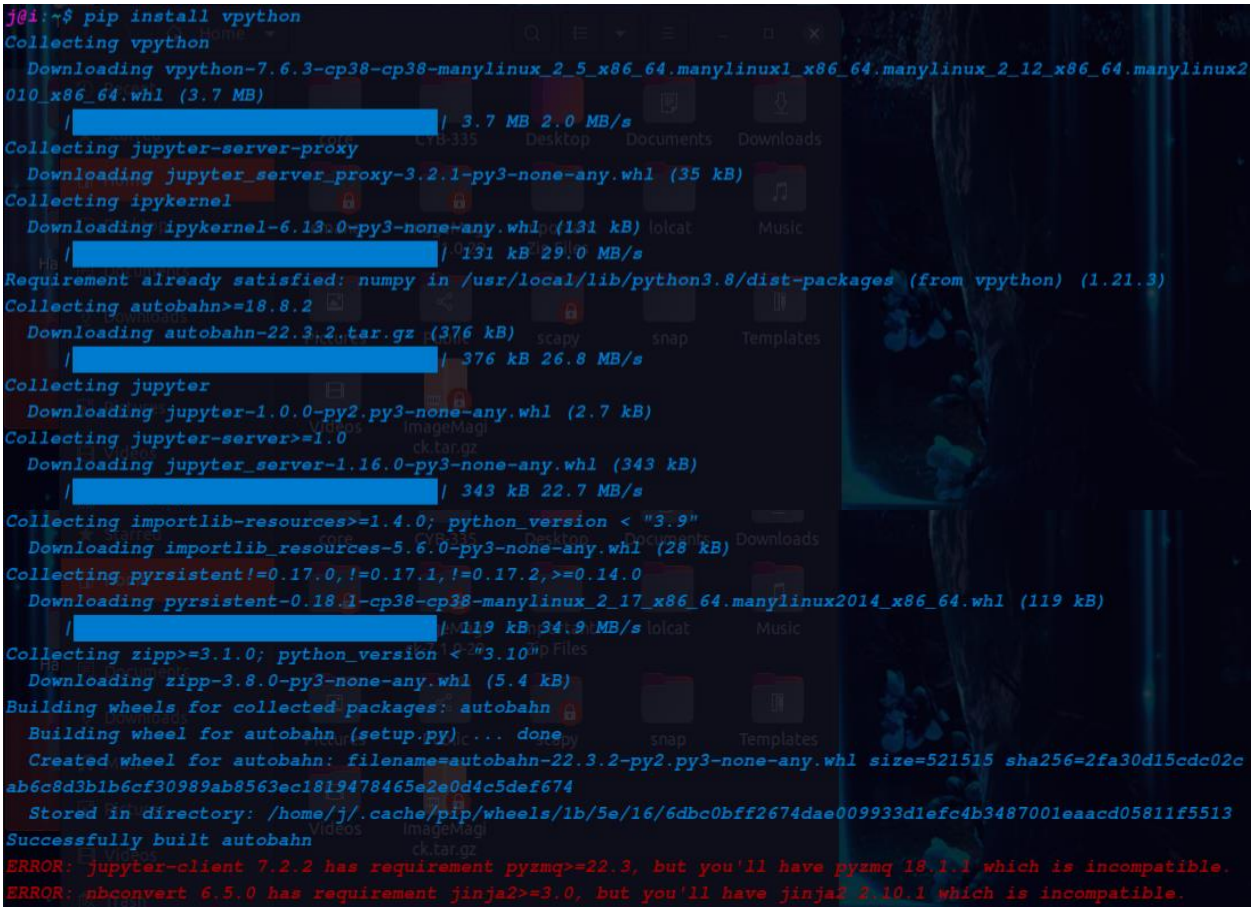

Figure 11 - vPython Install

To first install tpyzmq, I used the following three commands: “sudo pip install pyzmq” and “pip install --no-binary=:all: pyzmq” and “sudo apt-get install libzmq3-dev” (Figure 12).

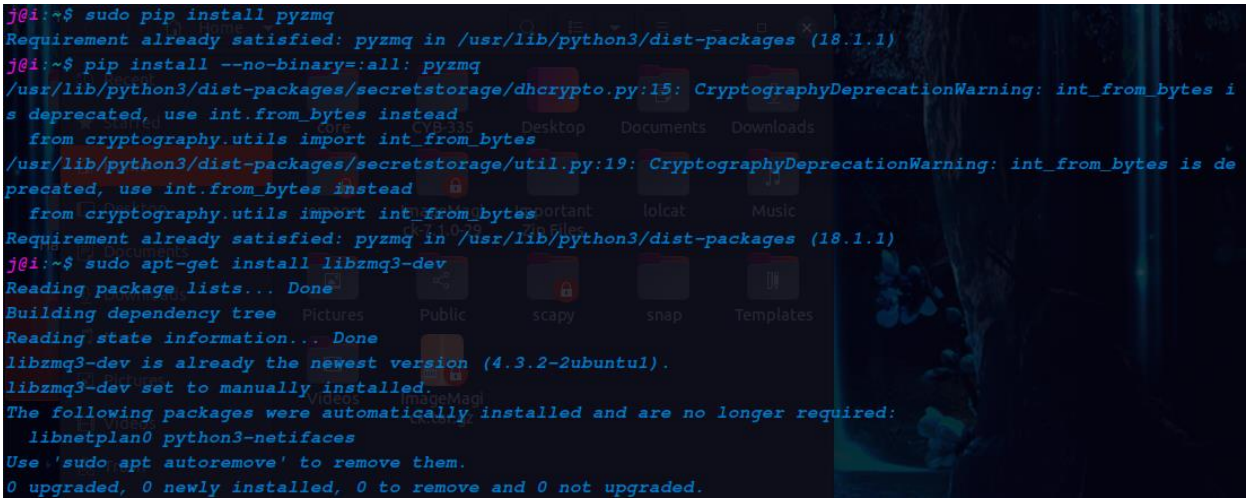

Figure 12 - pyzmq & libzmq3-dev Install

Now we can install the jinja Python module. To do this, I used the command “sudo apt-get install -y python-jinja2” (Figure 13).

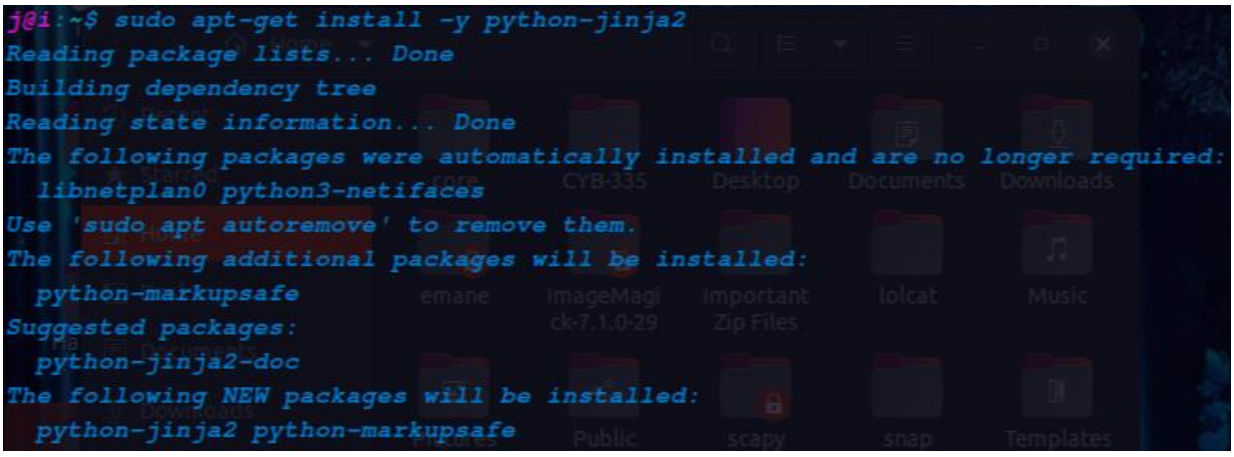

Figure 13 - Python-jinja2 Install

Now that these other dependencies for vPython are installed, I re-installed this by running the same command “pip install vpython” (Figure 14). This installed pyzmq and allowed vPython to successfully install (Figure 14).

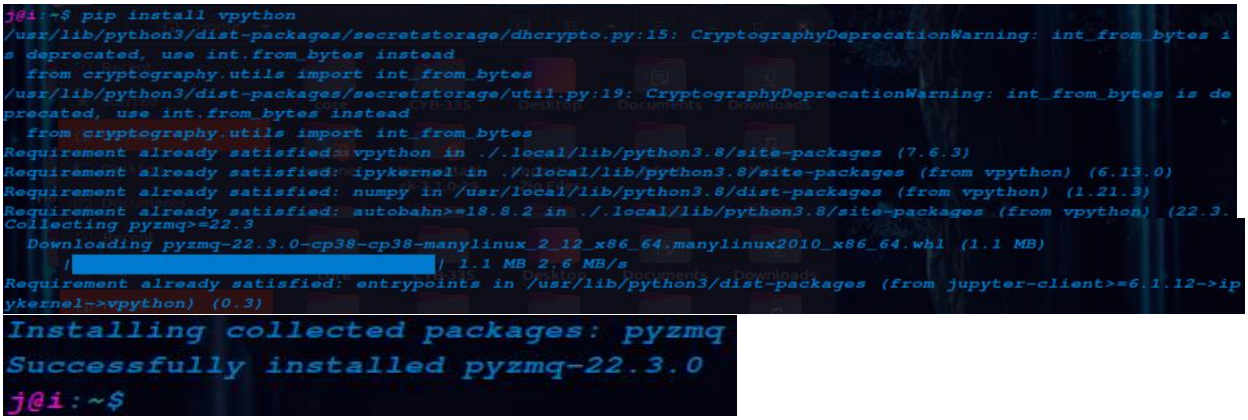

Figure 14 - Successful vPython Install

Cryptography, Nmap, OpenSSL, LibSSH2, Linssh2-1-dev & SoX

Now we can install Cryptography, Nmap, OpenSSL, LibSSH2, Linssh2-1-dev & SoX. These are more dependencies of Scapy. First I installed Cryptography using the command “sudo pip install cryptography” (Figure 15). I then needed to install snap and Nmap. To do this, I used the following commands: “sudo apt install nmap” & “sudo apt install snapd” & sudo snap install nmap” (Figure 15). Next I needed to install the source code for Nmap. To do this, I used the following commands: “sudo wget https://nmap.org/dist/nmap-7.91.tar.bz2” & “sudo tar xf nmap-7.91.tar.bz2” & “cd nmap-7.91/” & “sudo ./configure” (Figure 16).

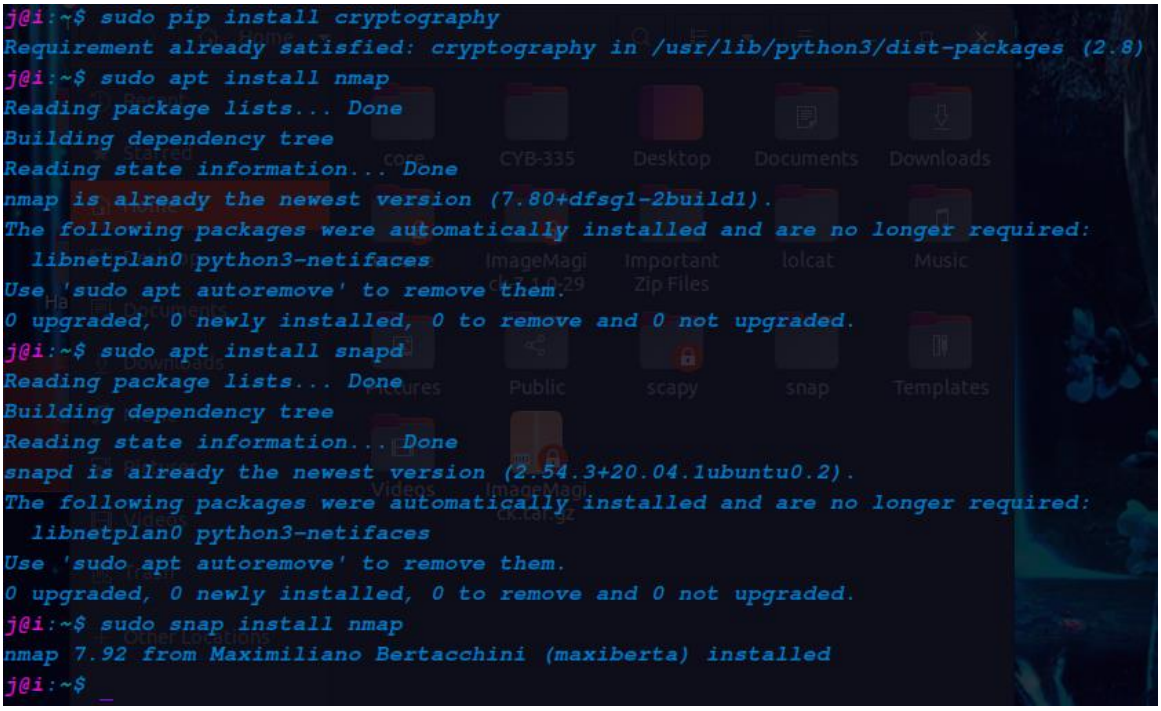

Figure 15 - Cryptography & Basic NMap Install

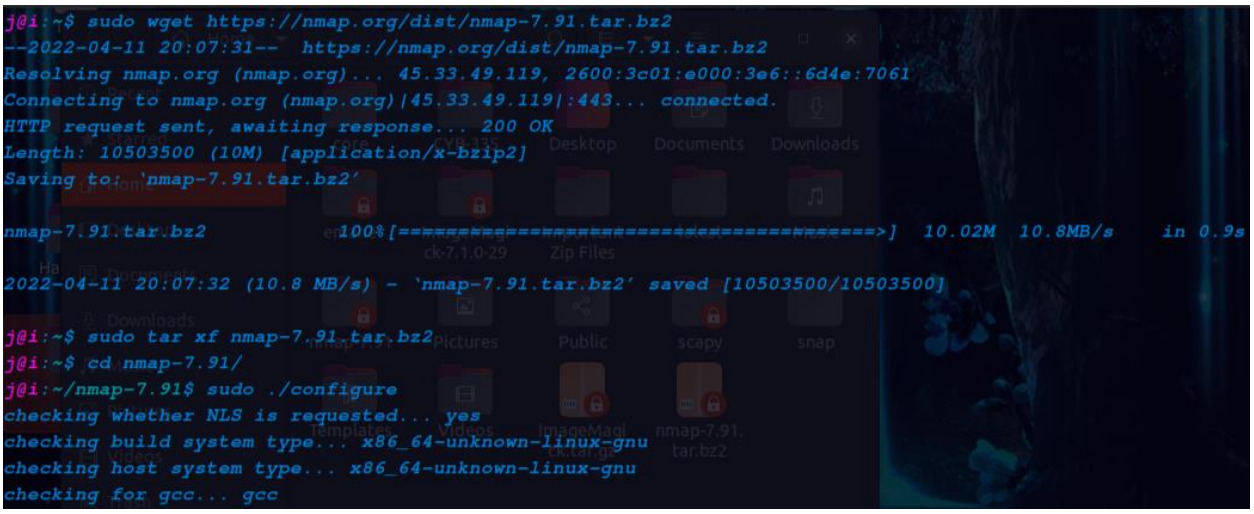

Figure 16 - NMap Install - Steps 1 - 4

After this was done, I noticed that two warnings appeared. They stated, “You are compiling without OpenSSL and LibSSH2” (Figure 17). So, I decided to install both of these.

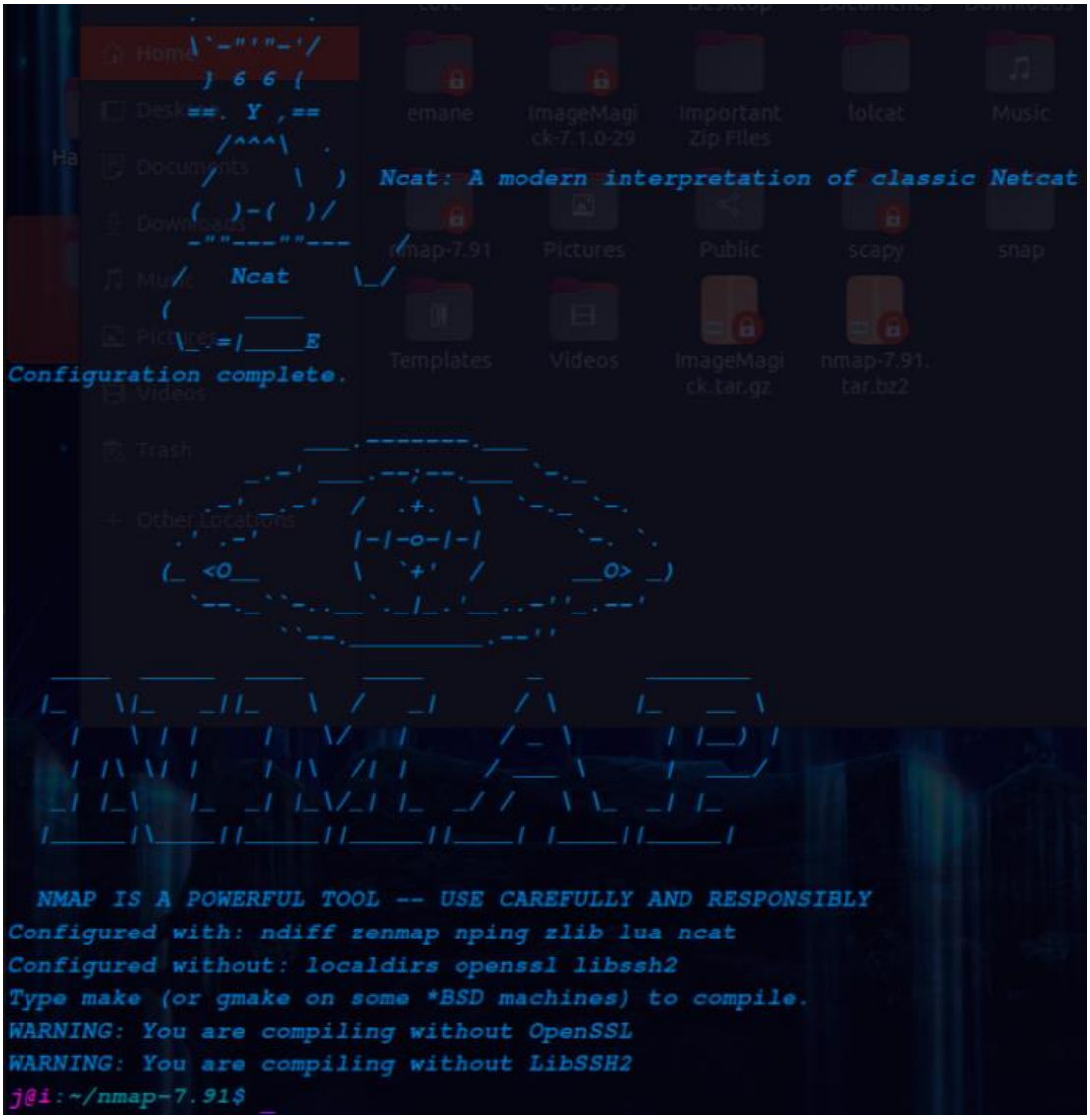

Figure 17 - NMap Successful Configuration

I first installed the Ubuntu packages “build-essential, checkinstall, and zlib1g-dev” (Figure 18). To do this, I used the following command: “sudo apt install build-essential checkinstall zlib1g-dev” (Figure 18). I then changed to the directory “/usr/local/src/” (Figure 19) and installed the source code for OpenSSL. To do this, I used the following commands:

- “sudo apt install build-essential checkinstall zlib1g-dev”
- “cd”
- “cd /usr/local/src”
- “sudo wget https://www.opensll.org/source/openssl-1.1.1k.tar.gz”

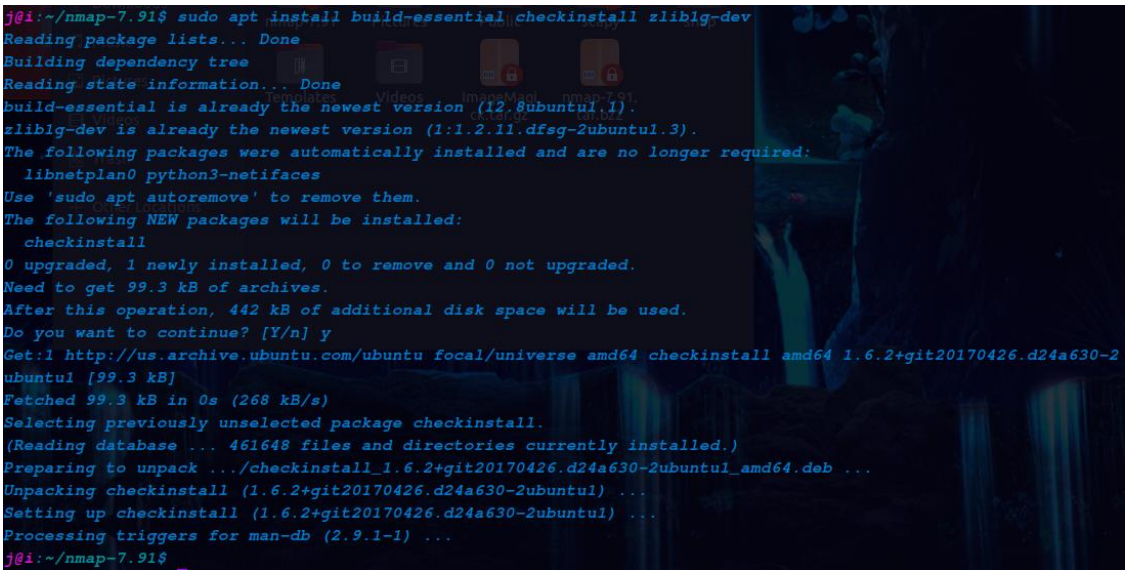

Figure 18 - NMap Install - Step 5

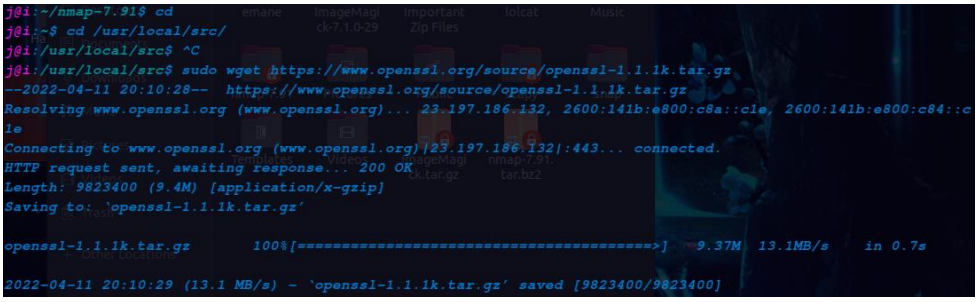

Figure 19 - OpenSSL Install - Steps 1 – 3

- “sudo tar -xf openssl-1.1.1k.tar.gz”
- “cd openssl-1.1.1k.tar.gz”
- “cd openssl-1.1.1k/”
- “sudo ./config --prefix=/usr/local/ssl --openssldir=/usr/local/ssl shared zlib”
- “sudo make && sudo make build && sudo make install”

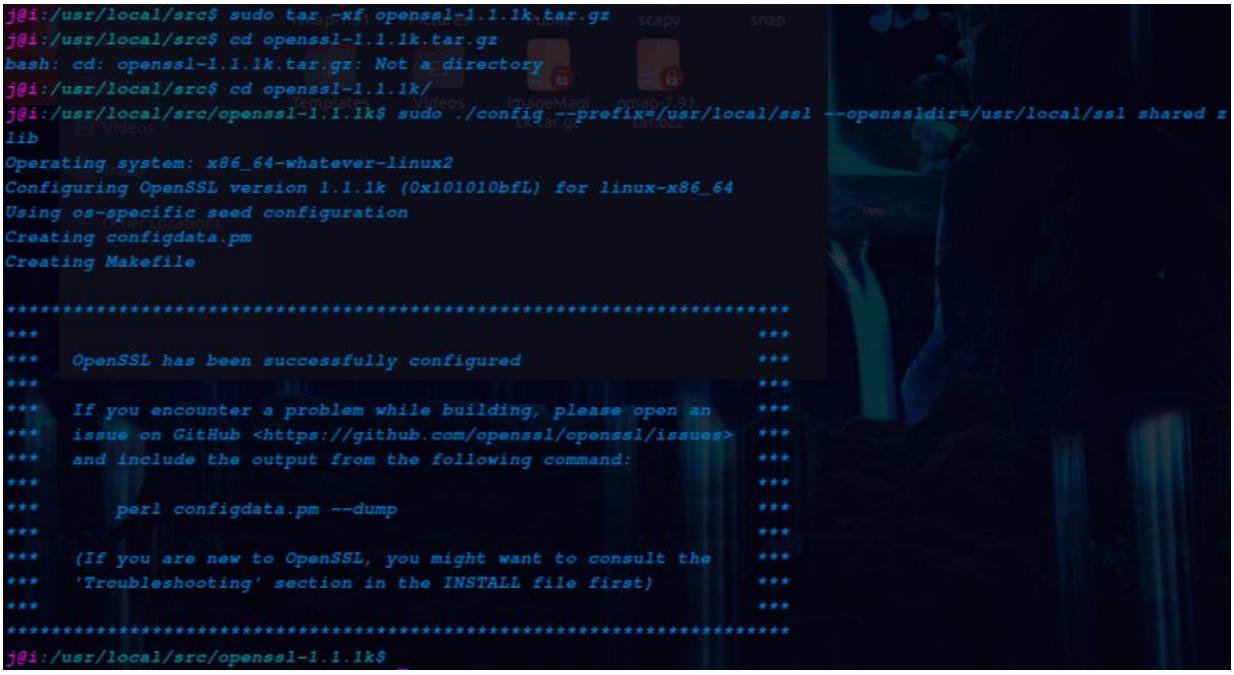

Figure 20 - OpenSSL Install - Steps 4 - 7

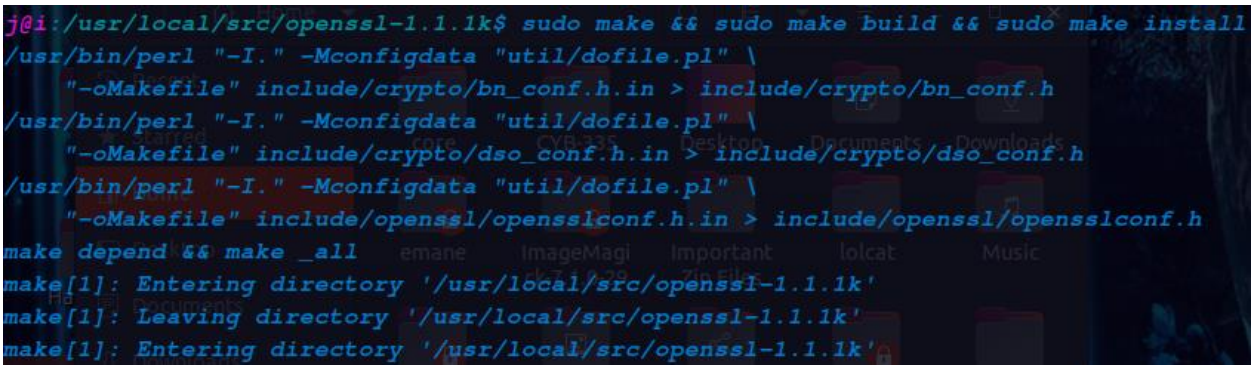

Figure 21 - OpenSSL Install - Step 8

The next package to install was LibSSH2, which can be installed using the command "sudo apt install libssh2-1-dev" (Figure 22). After installing these two packages, I changed back to the directory 'nmao-7.91' and used the command "sudo make" (Figure 23).

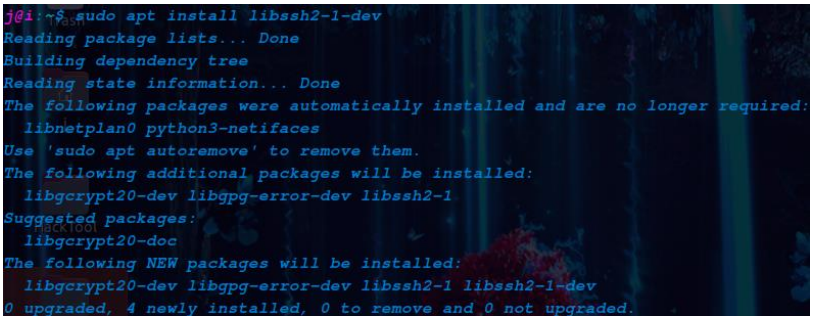

Figure 22 - Libssh2-1-dev Install

Figure 23 - NMap Install - Step 6

After this ran successfully, Nmap, OpenSSL, and LibSSH2 were installed successfully. So, the last dependency that is needed to install scapy is SoX. This cam be installed using the command "sudo apt install sox" (Figure 24).

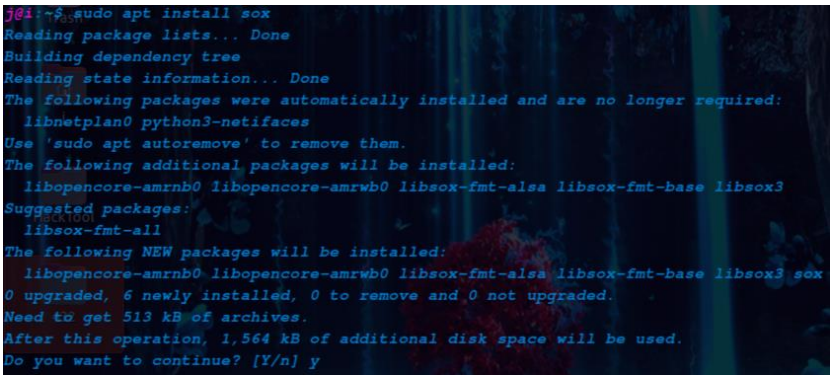

Figure 24 - SoX Install

Scapy Install Verification

Now that Scapy, and all of its required dependencies are installed, as well successfully, we can now run the command “scapy” within the terminal to start the program. After starting scapy, I configured Scapy to read pcap files by running the two commands: “from scapy.config import conf” & “conf.use_pcap = True” (Figure 25). Now that we have verified that Scapy is installed, we can start testing scapy out.

Figure 25 - Scapy Install Verification & Configuring Super-Sockets

Scapy Test

In this section, we will be following the interactive tutorial on the Scapy Docs. This is the 'Usage' section of the Scapy Documentation. It teaches people how to get use to Scapy's interactive shell, as well commands used by Scapy.

Customizing Terminal, Building Packet & Stacking Layers –

Before using the Scapy interface, they have different theme options that change the color of the interface. These consist of the "DefaultTheme, BrightTheme, RastaTheme, ColorOnBlackTheme, BlackAndWhite, HTMLTheme, LatexTheme" (Scapy, 2022). In this case, I choose the Rasta Theme as it was the easiest to view (Figure 26). After customizing the terminal theme, we can start the first steps of using scapy, which is to build a packet, and play around with it.

After completing the first steps, we can now start the next section, stacking layers (Scapy, 2022). Using the '/' operator represents the "composition operator between two layers" (Scapy, 2022). When using this operator, "the lower layer can have one or more of its defaults fields overloaded according to the upper layer" (Scapy, 2022). Any packet dissected still contans all filled data (fields). This is because we took into consideration "that each field has its value imposed by the original string" (Scapy, 2022).

The commands used to do this are listed above the section they are in, as well their output (Figure 26 - 28). Some commands have a long output, so these will not be seen listed with the command, however, can be seen in the Figure that corresponds with it. Example: ("command" = output) or ("command" & "command" = output).

First steps:

- “a=IP(ttl=10)” & “a” = < IP ttl=10 |>
- “a.src” = ’127.0.0.1’
- “a.dst="192.168.1.1"” & “a” = < IP ttl=10 dst=192.168.1.1 |>
- “a.src” = ’192.168.8.14’
- “del(a.ttl)” & “a” = < IP dst=192.168.1.1 |>
- “a.ttl” = 64

Stacking layers:

- “IP()” = <IP |>
- “IP()/TCP()” = <IP frag=0 proto=TCP |<TCP |>>
- “Ether()/IP()/TCP()” <Ether type=0x800 |<IP frag=0 proto=TCP |<TCP |>>>
- “IP()/TCP()/"GET / HTTP/1.0\r\n\r\n"” = <IP frag=0 proto=TCP |<TCP |<Raw load='GET / HTTP/1.0\r\n\r\n' |>>>

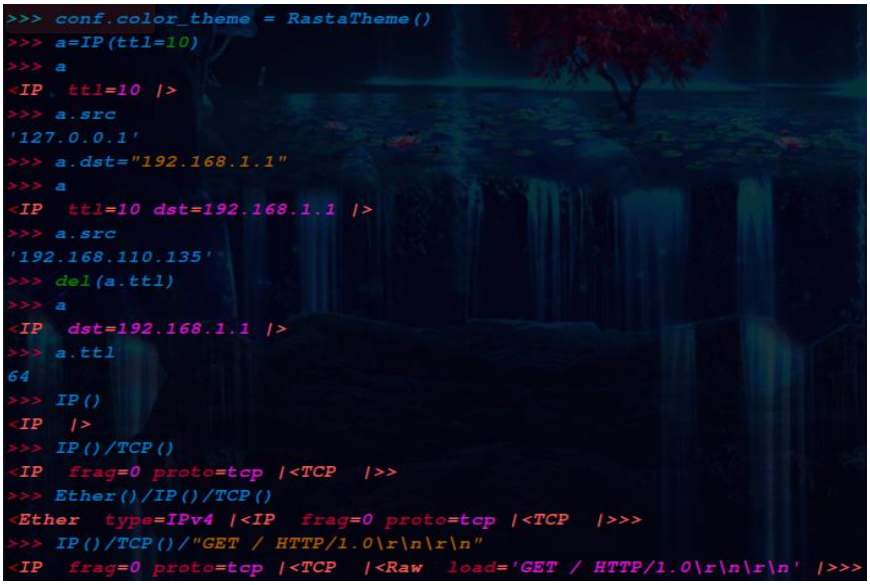

Figure 26 - Ubuntu VM Terminal - Scapy Test - 1

- “Ether()/IP()/IP()/UDP()” = <Ether type=0x800 |<IP frag=0 proto=IP |<IP frag=0 proto=UDP |<UDP |>>>>
- “IP(proto=55)/TCP()” = <IP frag=0 proto=55 |<TCP |>>
- ‘raw(IP())” = 'E\x00\x00\x14\x00\x01\x00\x00@\x00|\xe7\x7f\x00\x00\x01\x7f\x00\x00\x01'
- “IP(_)” = <IP version=4L ihl=5L tos=0x0 len=20 id=1 flags= frag=0L ttl=64 proto=IP chksum=0x7ce7 src=127.0.0.1 dst=127.0.0.1 |>
- “a=Ether()/IP(dst="www.slashdot.org")/TCP()/"GET /index.html HTTP/1.0 \n\n"” & “hexdump(a)” = (Figure 28).
- “b=raw(a)” & “b” = (Figure 28).
- “c=Ether(b)” & “c” = (Figure 28).
- “c.hide_defaults()” & “c” = (Figure 28).

```
>>> Ether()/IP()/IP()/UDP()
<Ether  type=IPv4 |<IP  frag=0 proto=ipencap |<IP  frag=0 proto=udp |<UDP  |>>>>
>>> IP(proto=55)/TCP()
<IP  frag=0 proto=55 |<TCP  |>>
>>> raw(IP())
b'E\x00\x00\x14\x00\x01\x00\x00@\x00|\xe7\x7f\x00\x00\x01\x7f\x00\x00\x01'
>>> IP(_)
<IP  version=4 ihl=5 tos=0x0 len=20 id=1 flags= frag=0 ttl=64 proto=hopopt chksum=0x7ce7 src=127.0.0.1 dst=12
7.0.0.1 |>
```

Figure 27 - Ubuntu VM Terminal - Scapy Test – 2

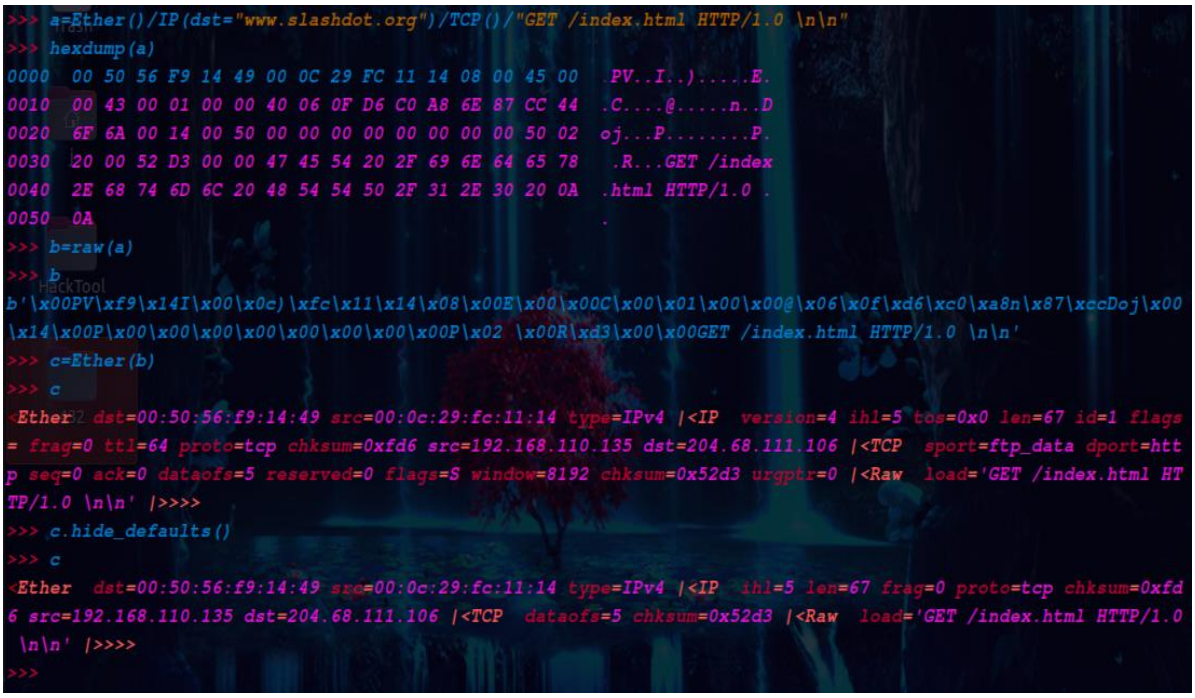

Figure 28 - Ubuntu VM Terminal - Scapy Test - 3

Reading PCAP Files & Graphical Dumps –

Scapy holds the ability to read packets from .pcap files, as well create and write to a .pcap file (Scapy, 2022). Within the file manager, I located the Scapy directory, selected the ‘test’ folder and selected the ‘pcaps’ folder (Figure 29). This is where some default Scapy pcap files reside (Figure 30). After examining some of these, I decided to use the file named “pfcp.pcap” which sends 200 UDP packets (Figure 30). Then we will create a graphical dump of data. If PyX is installed, users are able to create “a graphical PostScript/PDF dump of a packet or a list of packets” (Scapy, 2020). This allows the user to view data in a picture form (Figure 32). Once again, just as the last section, the command as well its output will be listed above its respected Figure, as well if the output is too long, the Figure that it resides in will replace the output.

- “a=rdpcap("/home/j/scapy/tests/pfcp.pcap")” & “a” = <pfcp.pcap: TCP:0 UDP:200 ICMP:0 Other:0>
- “a=rdpcap("/home/j/scapy/tests/doip_ack.pcap")” & “a” = <doip_ack.pcap: TCP:1 UDP:0 ICMP:0 Other:0>
- “a=rdpcap("/home/j/scapy/tests/http_comppressed.pcap")” & “a” = <http_comppressed.pcap: TCP:3 UDP:0 ICMP:0 Other:0>
- “a=rdpcap("/home/j/scapy/tests/http2_h2c.pcap")” & “a” = <http2_h2c.pcap: TCP:10 UDP:0 ICMP:0 Other:0>
- “a=rdpcap("/home/j/scapy/tests/ecu_trace.pcap.gz")” & “a” = <ecu_trace.pcap.gz: TCP:0 UDP:0 ICMP:0 Other:141>

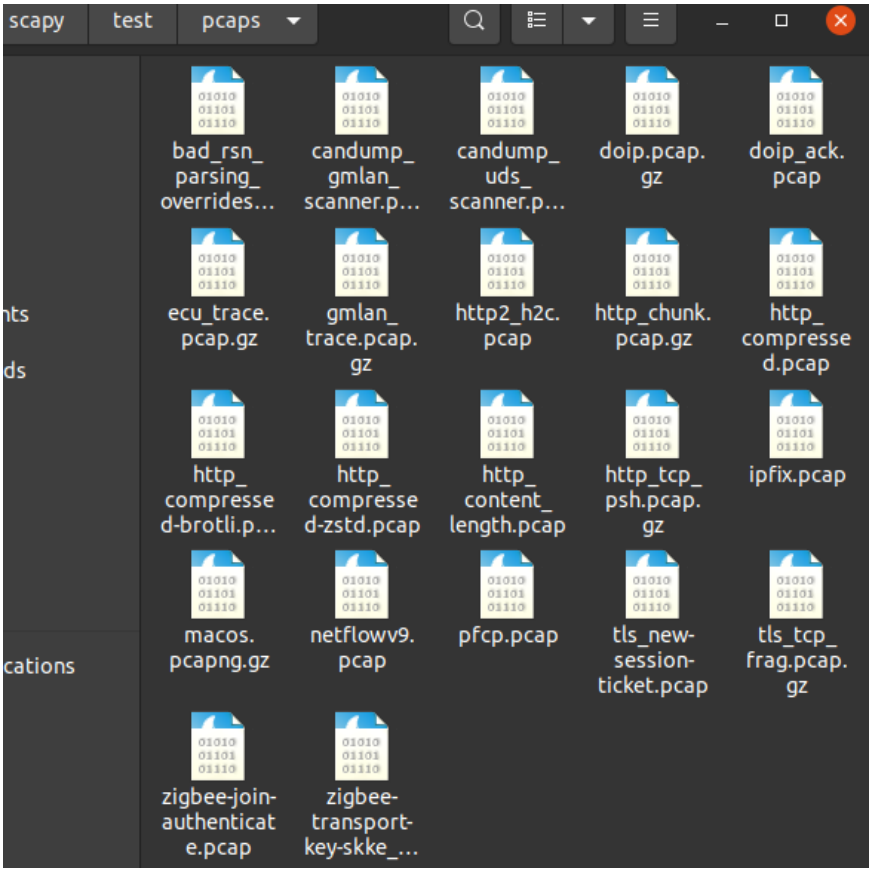

Figure 29 - Ubuntu VM Terminal - Scapy Test - 4

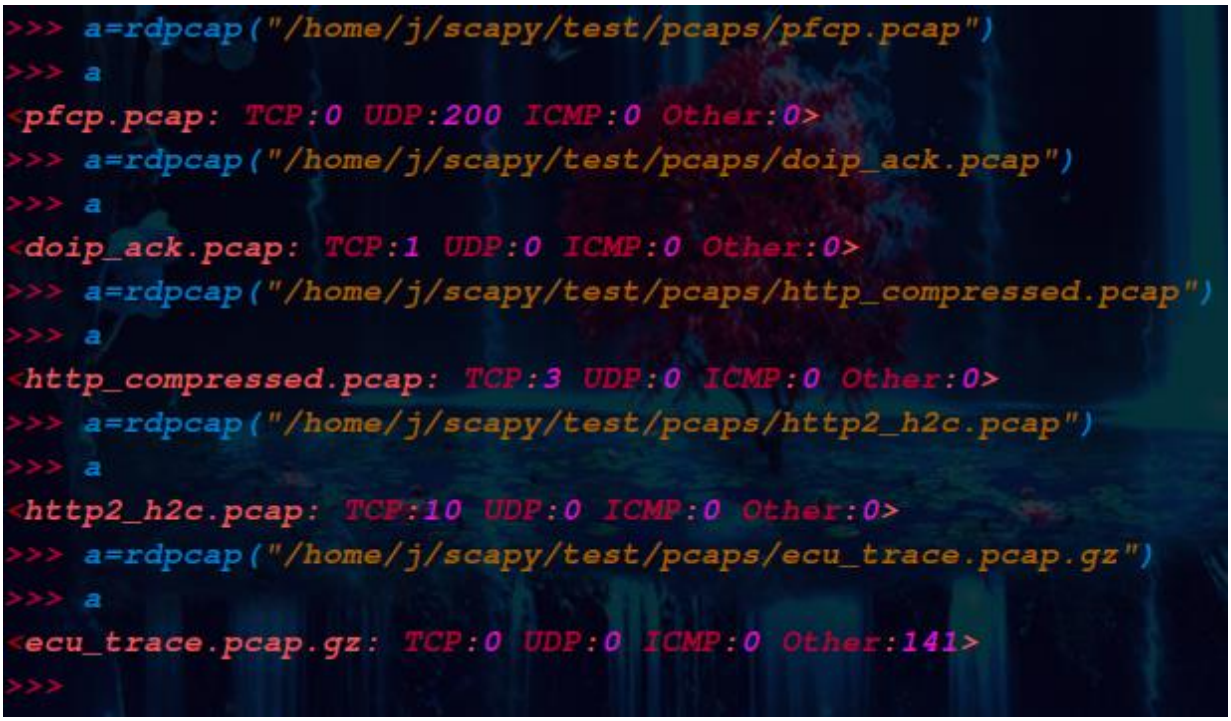

Figure 30 - Ubuntu VM Terminal - Scapy Test - 6

- "a[423].pdfdump(layer_shift=1)" = (Figure 31).

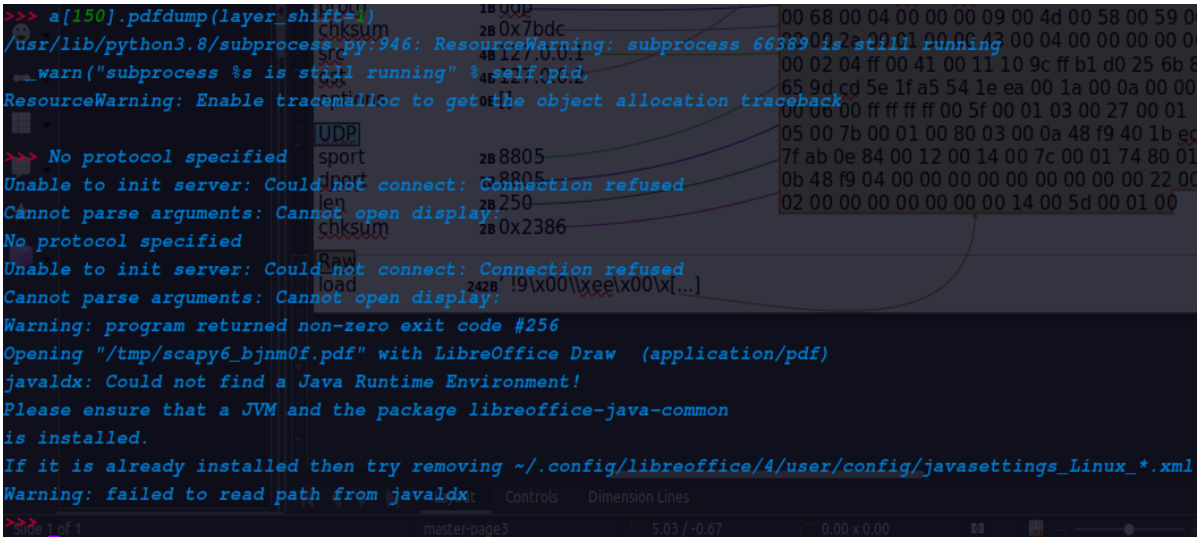

Figure 31 - Ubuntu VM Terminal - Scapy Test - 7

- Graphical Dump Created from previous command

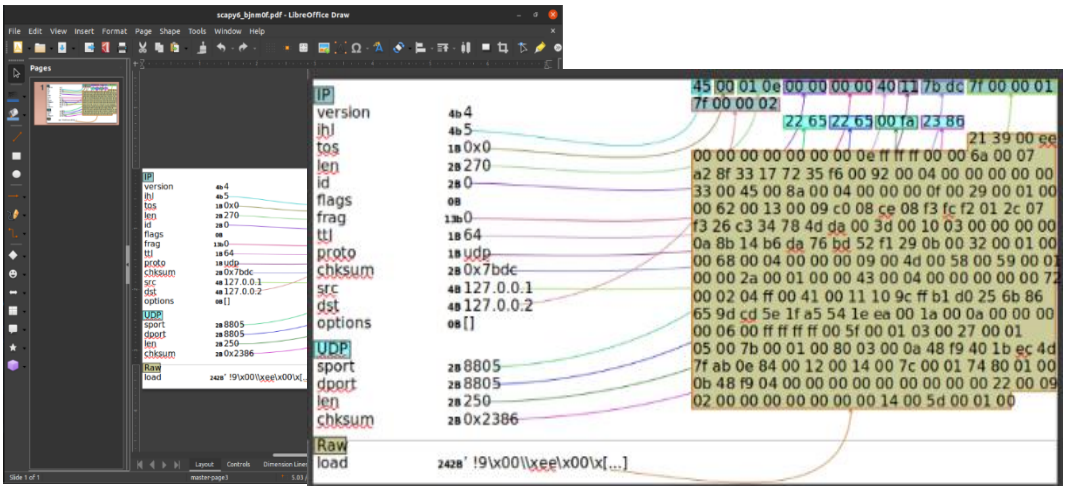

Figure 32 - Ubuntu VM Terminal - Scapy Test - 8

Generating Sets of Packets –

As of now, only one packet has been generated, however, in this section we will be generating sets of packets. Every field of an entire packet can be changed and configured (Scapy, 2022). “This implicitly defines a set of packets, generated using a kind of cartesian product between all the fields” (Scapy, 2022). A few operations will not “work on a set of packets” and this occurs when users forget to “unroll” the packet set (Scapy, 2022). The first element of the list, and only this, when not generated by the user, “will be used to assemble the packet” (Scapy, 2022).

- “a=IP(dst="www.slashdot.org/30")” & “a” = <IP dst=Net('www.slashdot.org/30') |>
- “[p for p in a]” = [<IP dst=204.68.111.104 |>, <IP dst=204.68.111.105 |>, <IP dst=204.68.111.106 |>, <IP dst=204.68.111.107 |>]
- “b=IP(ttl=[1,2,(5,9)])” & “b” = <IP ttl=[1, 2, (5, 9)] |>
- “[p for p in b]” = [<IP ttl=1 |>, <IP ttl=2 |>, <IP ttl=5 |>, <IP ttl=6 |>, <IP ttl=7 |>, <IP ttl=8 |>, <IP ttl=9 |>]
- “c=TCP(dport=[80,443])” & “[p for p in a/c]” = (Figure 33).

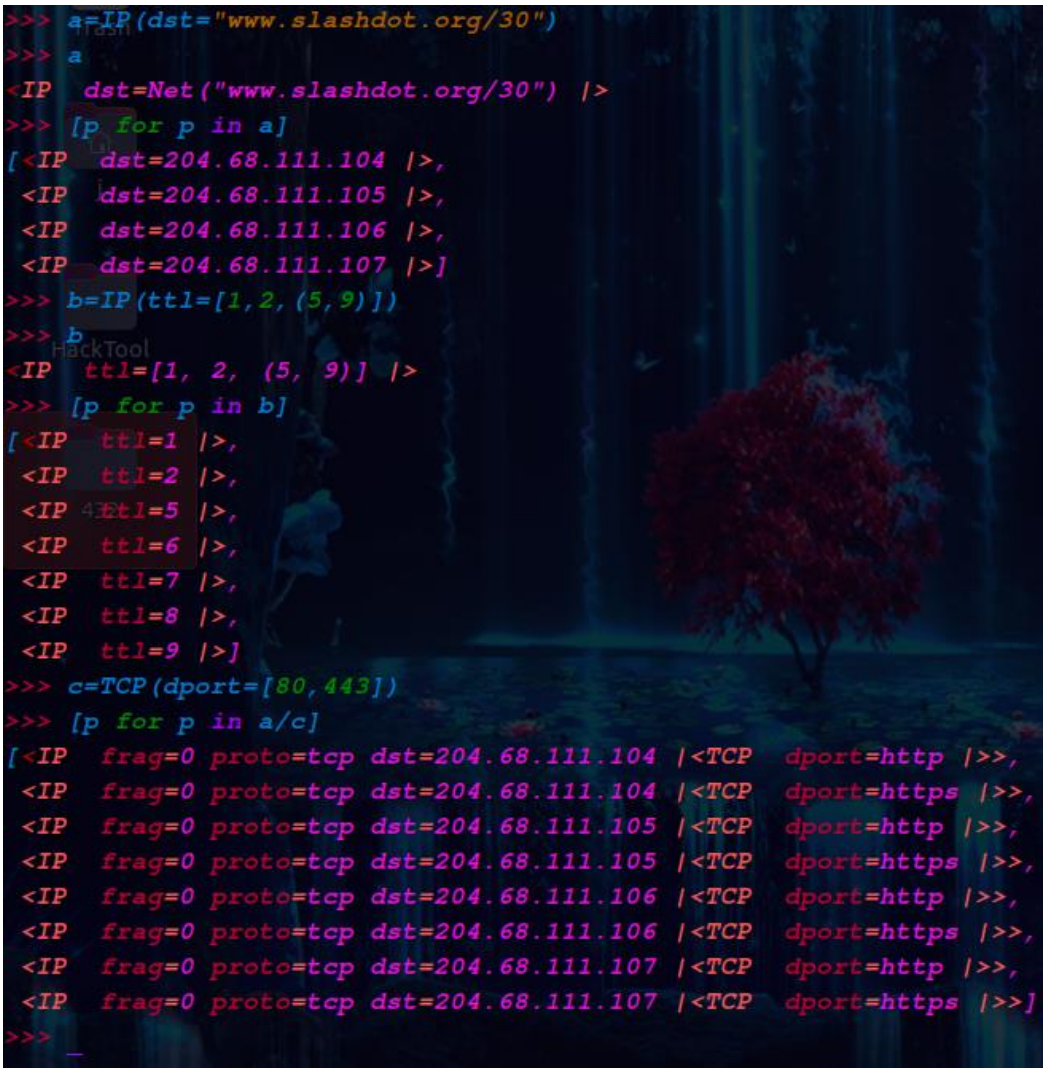

Figure 33 - Ubuntu VM Terminal - Scapy Test - 9

CORE Switch Network Topology

After completing the previous section and moving on to the Sending Packets section in the practice run, I noticed that these commands worked on the Ubuntu VM terminal, but I wanted to see what commands could be sniffed by TCPDump and what ones couldn't. So, for the purpose of this, in this run I decided to create a basic switched environment, however for some reason without realizing it until afte completing the entire lab that I used a Hub on accident, so for time sake we will be continuing with a CORE network topology using three nodes (PCs) and one Hub (Figure 34). After placing the nodes, I linked them to the hub. After doing this, I right-clicked each PC and renamed them: PC1, PC2, and PC3-L (Figure 34). PC1 and PC2 will be used to communicate with each other, and PC3-L will be listening to them communicate by sniffing the traffic with TCPDump (Figure 34). As well, I obtained the IP address for these nodes, which are listed below, as well can be seen in Figure 34. After obtaining these IPs, I started the CORE emulation by selecting the green play button (Figure 34).

- PC1: 10.0.0.20
- PC2: 10.0.0.21
- PC3-L: 10.0.0.22

After starting the emulation, to be sure that it is set up correctly, I double-clicked each node to open the terminal (Figure 35). This allows the user to run commands as if they were within the Ubuntu VM terminal, as well allows the user access to anything installed in their Ubuntu VM. After the terminal opened, I used the ping command to ping the other nodes (Figure 35). After this, I used 'ifconfig' to view network interface information to gain the default interface. In this case, the interfaced used was 'eth0' (Figure 36).This was successful, the commands used to do this will be listed above their respected Figure.

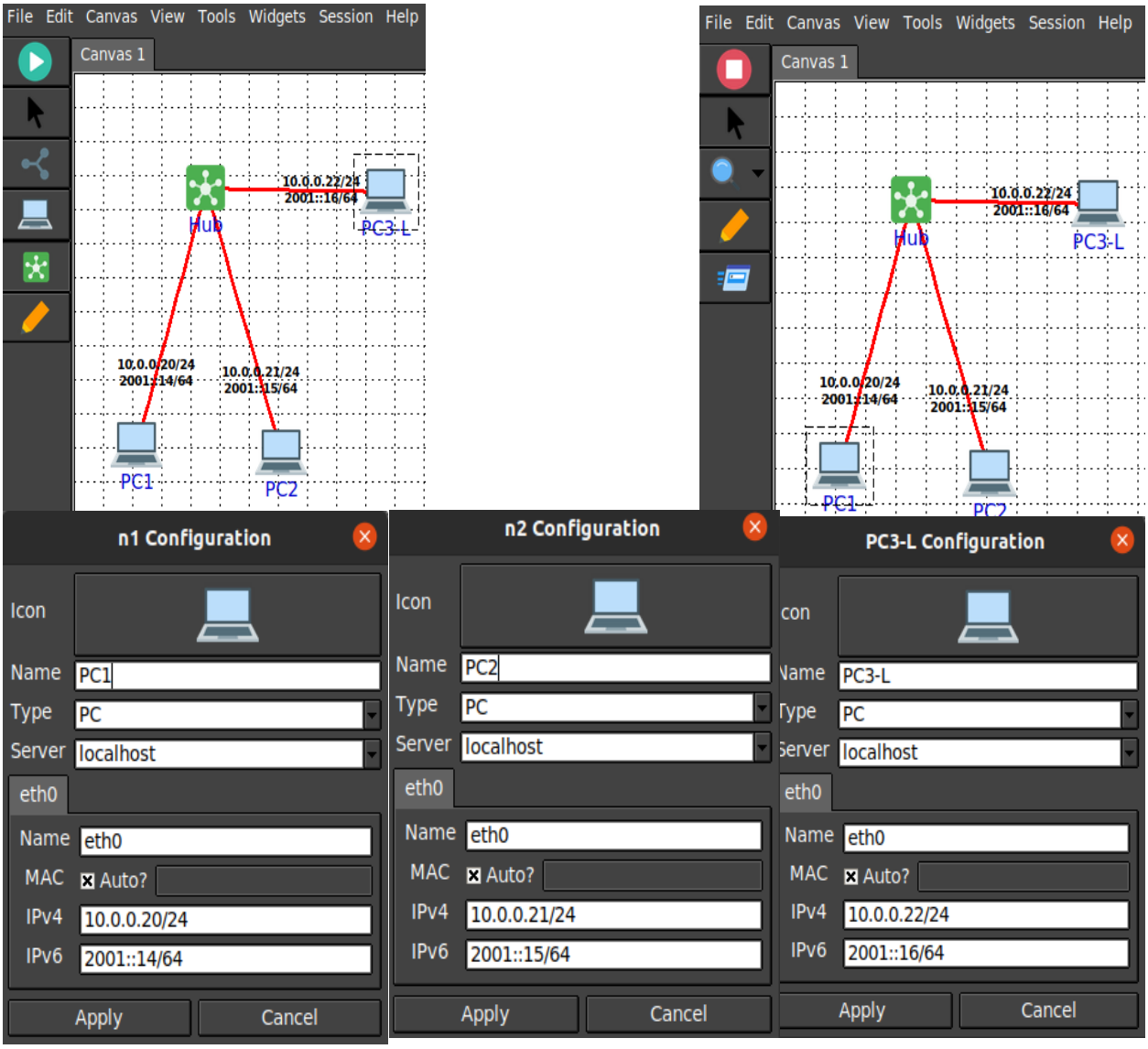

Figure 34 - CORE Network Topology - Steps 1 - 5

- PC1: “ping 10.0.0.21” & “ping 10.0.0.22” (Figure 35)
- PC2: “ping 10.0.0.20” & “ping 10.0.0.22” (Figure 35)
- PC3-L: “ping 10.0.0.20” & “ping 10.0.0.21” (Figure 36)
- “ifconfig” (Figure 36)

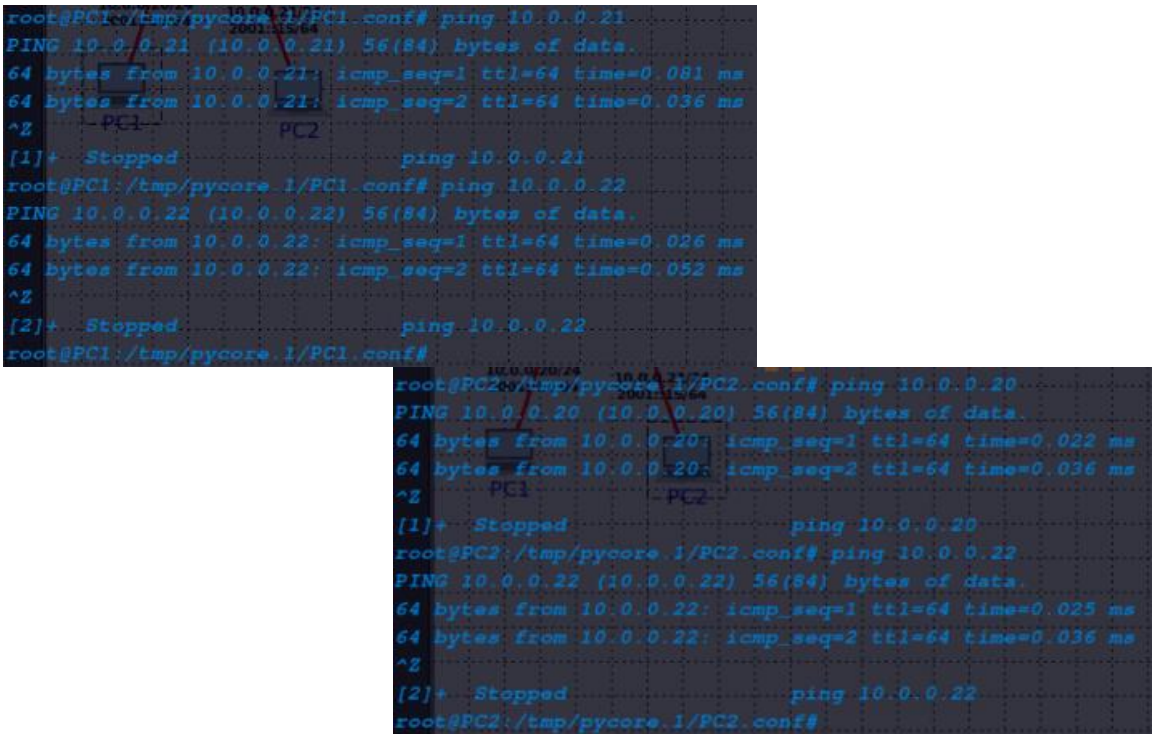

Figure 35 - CORE Network Topology - Steps 6 - 7

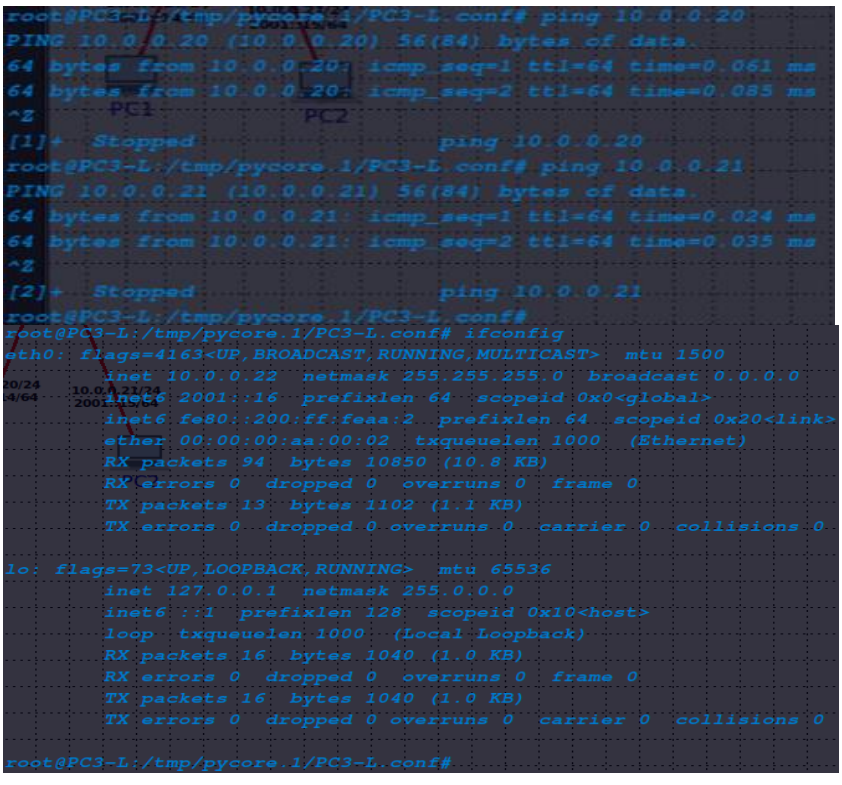

Figure 36 - CORE Network Topology - Steps 8 - 9

After successfully verifying that the CORE network topology is set-up successfully, using the command “sudo scapy” within the terminal of node PC1, we can access to Scapy interface and continue with the interactive tutorial in the Scapy documentation (Figure 37).

Figure 37 - CORE Network Topology - Steps 10

Scapy Examples & Results

In this section, we will be continuing from where we left off previously when working on the Scapy interactive tutorial, which was Generating Sets of Packets. The next section is Sending Packets. This section will also cover Fuzzing, Injecting Bytes, Send and Receive Packets, SYN Scans, and finally TCP Traceroute (Scapy, 2022). This section will be laid out the same as the other interactive tutorial section.

Sending Packets

After learning how to manipulate packets, we can now learn how to send them (Scapy, 2022). Using the send() function allows the user to “send packets at layer 3” (Scapy, 2022). It will also “handle routing and layer 2” (Scapy, 2022). Using the sendp() function also works at layer 2 however it’s up to the user to select the correct interface and layer protocol (Scapy, 2022). “send() and sendp() will also return sent packet list if return_packets=True is passed as parameter” (Scapy, 2022). The commands used are listed above their respected figure.

- “send(IP(dst="10.0.0.21")/ICMP())” = . Sent 1 packets.
- “sendp(Ether()/IP(dst="10.0.0.21",ttl=(1,4)), iface="eth1")” = ... Sent 4 packets.
- “sendp("I'm travelling on Ethernet", iface="eth0", loop=1, inter=0.2)” = ...^C Sent 18 packets.
- “sendp(rdpcap("/home/j/scapy/test/pcaps/pfcp.pcap"))” = Sent 200 packets.

Figure 38 - Scapy Docs Usage Tutorial - Sending Packets – 1

After running the previous commands, I noticed that TCPDump, running in the terminal of PC3-L had sniffed some traffic. This contained a mass amount of data, which means the output takes up a few figures. Most of the data here is ICMP traffic, I did see some payload traffic as well (Figure 39). There was also some 'who-has' requests that I noticed (Figure 40). To my surprise, TCPDump captured traffic from an unknown source (Figures 41 & 42). The data in these can be viewed by zooming in this document.

- "tcpdump -v"

Figure 39 - Scapy Docs Usage Tutorial - Sending Packets - 2

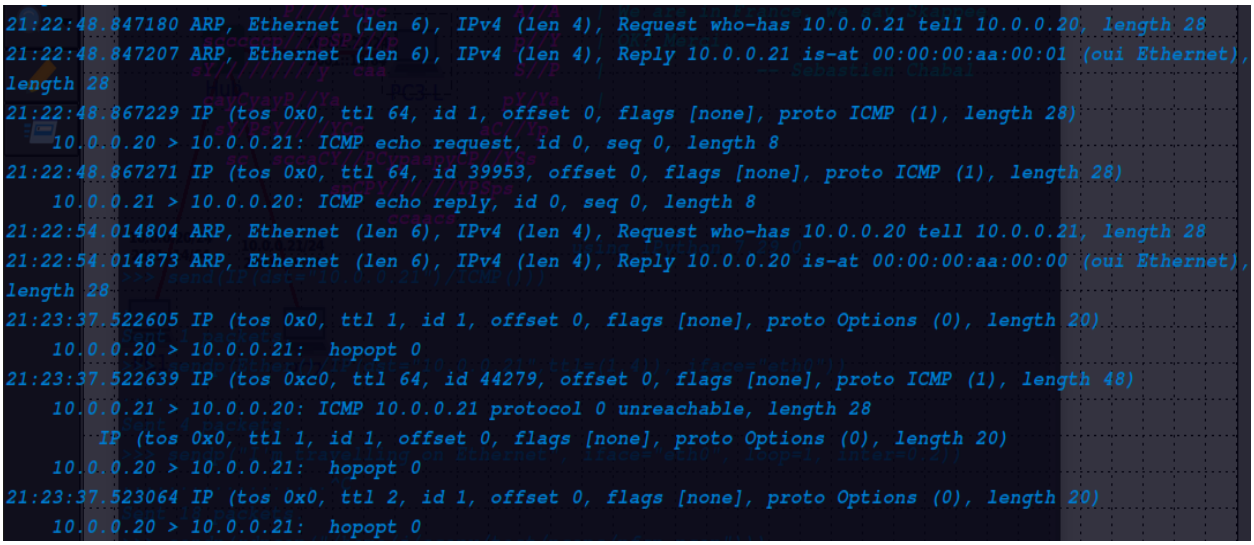

Figure 40 - Scapy Docs Usage Tutorial - Sending Packets - 3

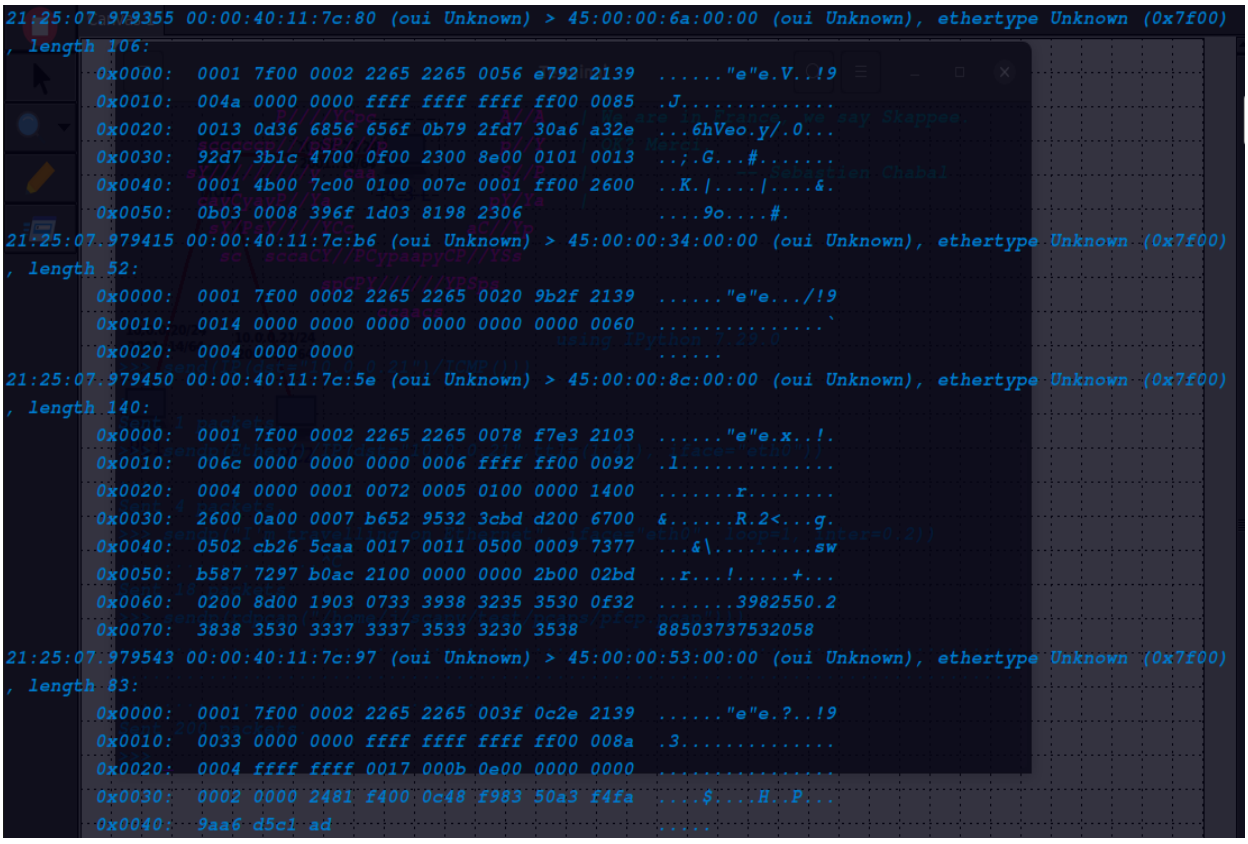

Figure 41 - Scapy Docs Usage Tutorial - Sending Packets - 4

Figure 42 - Scapy Docs Usage Tutorial - Sending Packets - 5

The last command from the Sending Packets section:

- "send(IP(dst='127.0.0.1'), return_packets=True)" = . Sent 1 packets. <PacketList: TCP:0 UDP:0 ICMP:0 Other:1>

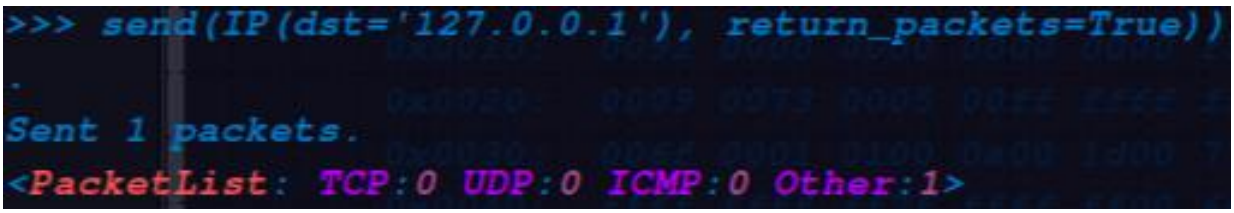

Figure 43 - Scapy Docs Usage Tutorial - Sending Packets - 6

Fuzzing

Using the fuzz() function allows the user the ability “to change any default value that is not to be calculated by an object whose value is random and whose type is adapted to the field” (Scapy, 2022). This works by enabling quick “building fuzzing templates and sending them in a loop” (Scapy, 2022).

- “send(IP(dst="10.0.0.21")/fuzz(UDP()/NTP(version=4)),loop=1) = …^C Send 129 Packets

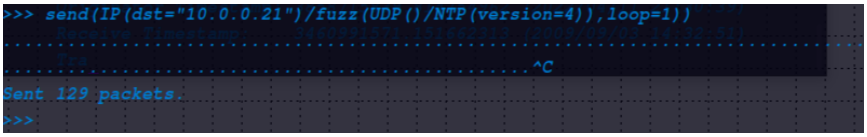

Figure 44 - Scapy Docs Usage Tutorial - Fuzzing - 1

This command also had traffic sniffed by TCPDump. The output contains who-has requests, as well replays (Figure 45 & 46). As well, it contains timestamp outputs (Figure 45 & 46). The command used to capture this was “tcpdump -v” (Figure 45). Again, to view this Figures it may be best to zoom this document in.

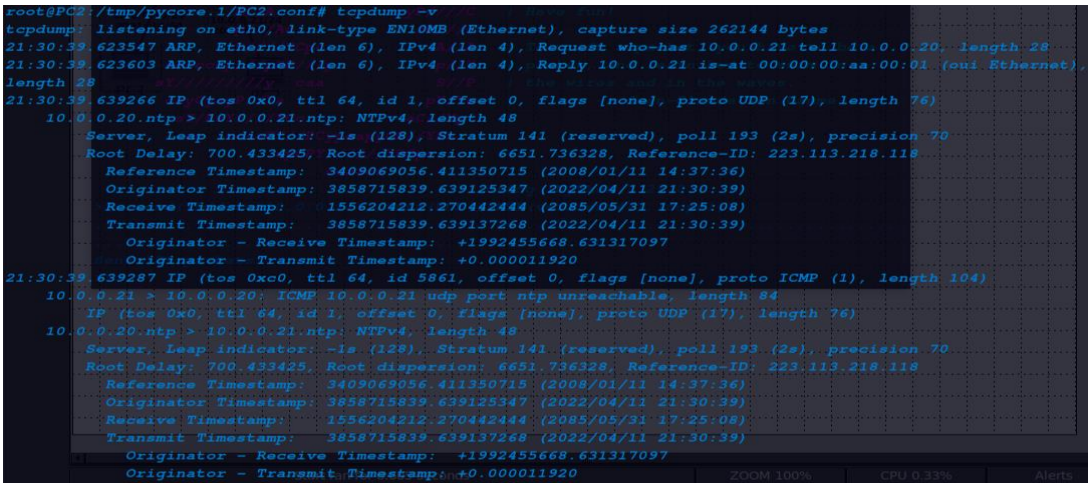

Figure 45 - Scapy Docs Usage Tutorial - Fuzzing - 2

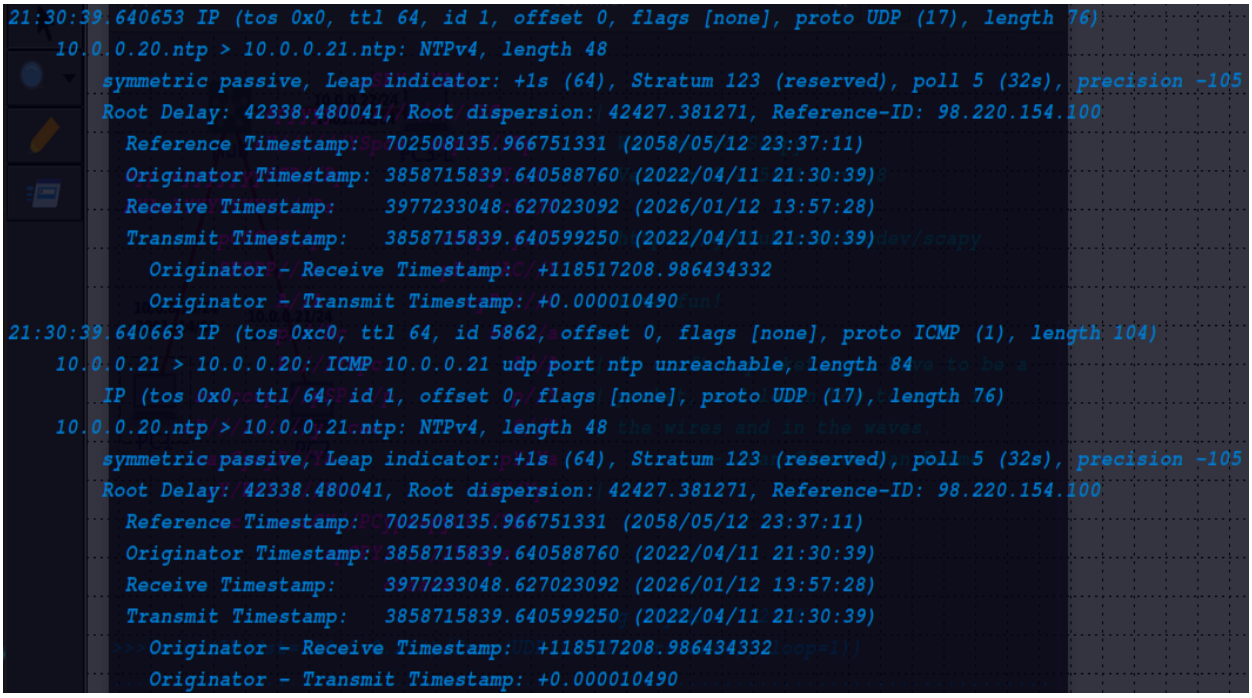

Figure 46 - Scapy Docs Usage Tutorial - Fuzzing - 3

Injecting Bytes

Now we can inject bytes into packets. Each packets have fields have a specific type (Scapy, 2022). An example is that “the length field of the IP packet ‘len’ expects an integer. More on that later. If you’re developing a PoC, there are times where you’ll want to inject some value that doesn’t fit that type. This is possible using RawVal” (Scapy, 2022).

- “pkt = IP(len=RawVal(b"NotAnInteger"), src="127.0.0.1")” & “bytes(pkt)” = b'H\x00NotAnInt\x0f\xb3er\x00\x01\x00\x00@\x00\x00\x00\x7f\x00\x00\x01\x7f\x00\x00\x01\x00\x00'

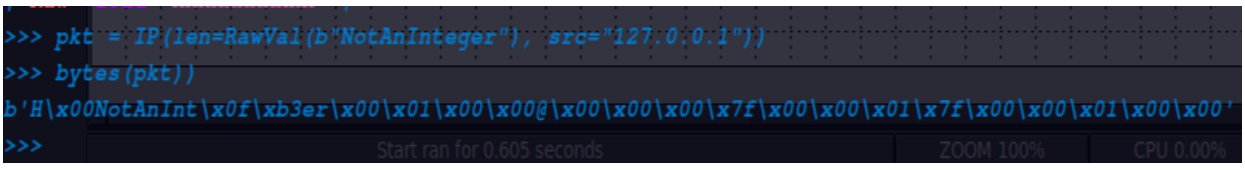

Figure 47 - Scapy Docs Usage Tutorial - Injecting Bytes – 1

Send & Receive Packets

Now we can try to send and receive packets using Scapy. Using the sr() function allows the user to send packets as well receive the answers (Scapy, 2022). This “The function returns a couple of packet and answers, and the unanswered packets” (Scapy, 2022). Using the sr1(), a version of sr(), which “only returns one packet that answered the packet (or the packet set) sent” as well these “packets must be layer 3 packets” (Scapy, 2022). Using the srp() function does this same thing except “for layer 2 packets (Ethernet, 802.3, etc.). If there is no response, a None-value will be assigned instead when the timeout is reached” (Scapy, 2022).

Send and Receive functions are the main part of Scapy, and they “return a couple of two lists. The first element is a list of couples (packet sent, answer), and the second element is the list of unanswered packets. These two elements are lists, but they are wrapped by an object to present them better, and to provide them with some methods that do most frequently needed actions:” (Figure 49) (Scapy, 2022).

When a “limited rate of answers” is returned, the user “can specify a time interval (in seconds) to wait between two packets with the inter parameter” (Scapy, 2022). Some packets may be lost if the interval is not enough , however users “can resend all the unanswered packets, either by calling the function again, directly with the unanswered list, or by specifying a retry parameter” (Scapy, 2022). As well, “he timeout parameter specify the time to wait after the last packet has been sent:” (Figure 50) (Scapy, 2022).

- “p = sr1(IP(dst="10.0.0.21")/ICMP()/"XXXXXXXXXXX")” = Begin emission: ... Finished to send 1 packets. .* Received 2 packets, got 1 answers, remaining 0 packets
- “p” & “p.show()” = (Figure 48).

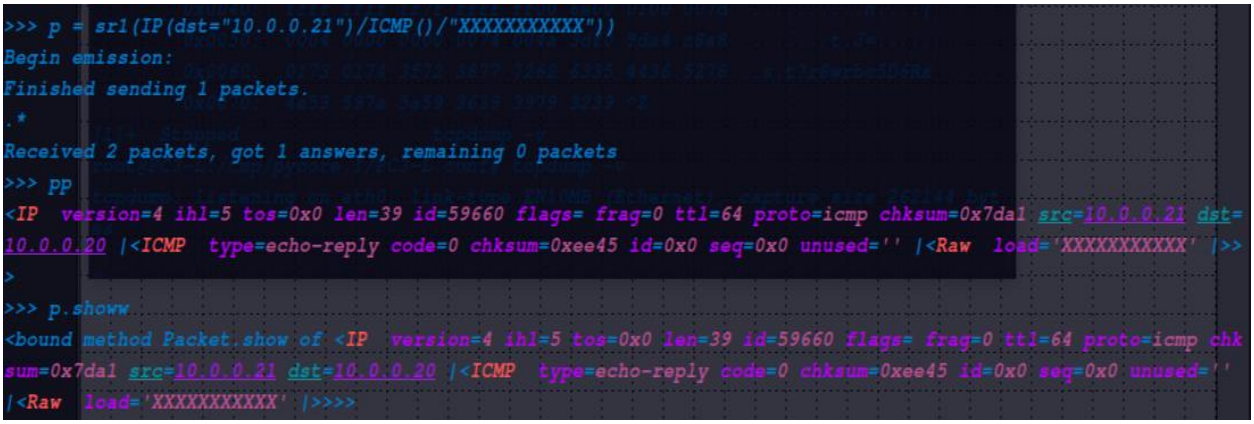

Figure 48 - Scapy Docs Usage Tutorial - Send & Receive Packets - 1

“A DNS query (rd = recursion desired)” (Scapy, 2022).

- “sr1(IP(dst="10.0.0.21")/UDP()/DNS(rd=1,qd=DNSQR(qname="www.slashdot.org")))” = (Figure 49).
- “sr(IP(dst="10.0.0.21")/TCP(dport=[21,22,23]))” = (Figure 49).
- “ans, unans = _” = (Figure 49).
- “ans.summary()” = (Figure 49).

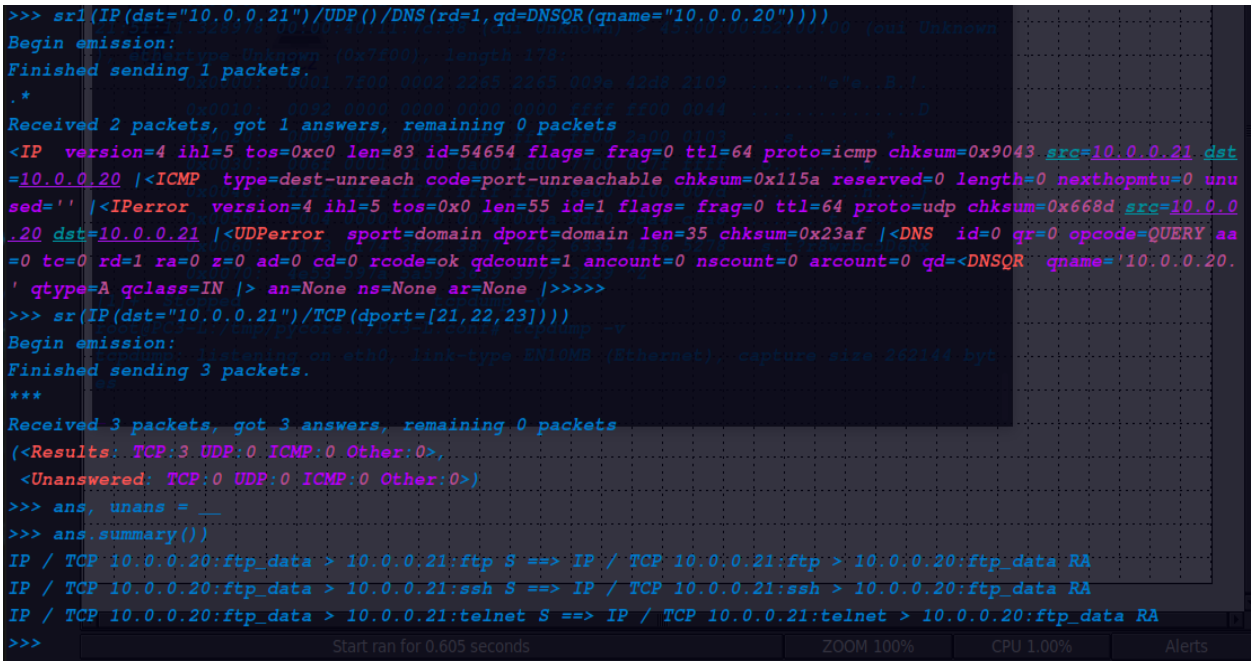

Figure 49 - Scapy Docs Usage Tutorial - Send & Receive Packets - 2

- "sr(IP(dst="172.20.29.5/30")/TCP(dport=[21,22,23]),inter=0.5,retry=-2,timeout=1)" = (Figure 50).

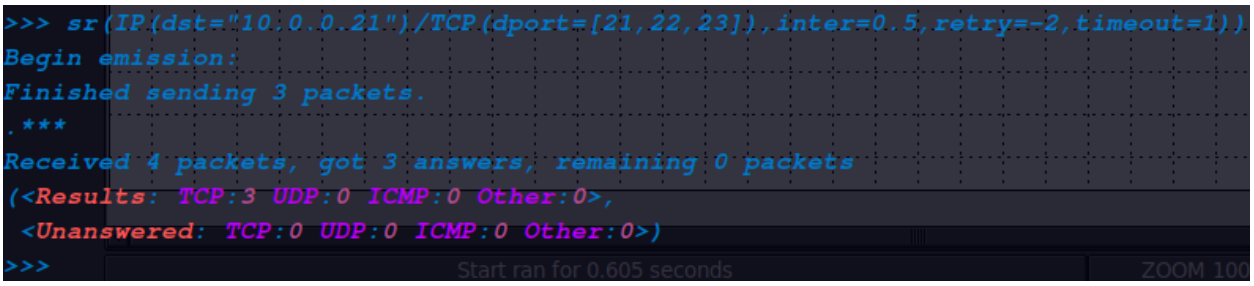

Figure 50 - Scapy Docs Usage Tutorial - Send & Receive Packets - 3

SYN Scans

The classic Scapy “SYN Scan can be initialized by executing” the command seen in Figure 51. This “will send a single SYN packet to” PC2 on “port 80 and will quit after receiving a single response” (Scapy, 2022). The output from this shows that PC2 “returned “RA” flag indicating an open port” and scans “ports 400 through 443 on the system” (Figure 51) (Scapy, 2022). We can view “responses simply request a summary of collected packets:” (Figure 54) (Scapy, 2022). This can be shown as “stimulus/response pairs for answered probes” and “can display only the information we are interested in by using a simple loop:” (Figure 54). We can even build a table “using the make_table() function to display information about multiple targets” (Figure 57). The next two examples after this did not work, however it “will even print the ICMP error type if the ICMP packet was received as a response instead of expected TCP” (Figure 57) (Scapy, 2022). When preforming large scans, we can choose to only display “certain responses” (Figure 57) (Scapy, 2022). There is also an option to perform an analysis on the responses, like indicating “which ports are open: Again, for larger scans we can build a table of open ports:” (Figure 57) (Scapy, 2022). If these methods don’t work or are “not enough, Scapy includes a report_ports() function which not only automates the SYN scan, but also produces a LaTeX output with collected results:” (Figure 57) (Scapy, 2022). The commands used as well their outputs will be listed above their repected Figures, and the same goes for traffic sniffed by TCPDump running in PC3-L.

- “sr1(IP(dst="10.0.0.21")/TCP(dport=80,flags="S")) = (Figure 51).
- “sr(IP(dst="10.0.0.21")/TCP(sport=666,dport=(440,443),flags="S"))” = (Figure 51).

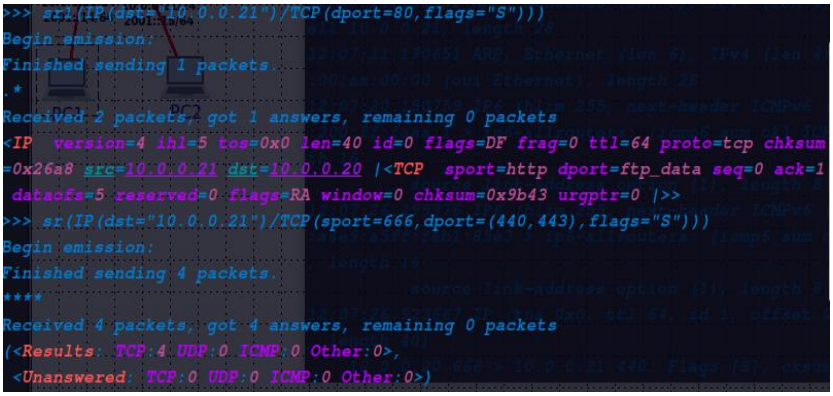

Figure 51 - Scapy Docs Usage Tutorial - SYN Scans - 1

On PC3-L using TCPDump, I was able to capture traffic after running the two previous commands. This consisted of ICMP packets as well payloads, router solicitation packets, and IPv6 payload packets (Figure 52). Then I saw who-has request/replay packets and messages stating “correct, none, DF” (Figure 53). To view the Figures this document should be zoomed in.

- “tcpdump -v”

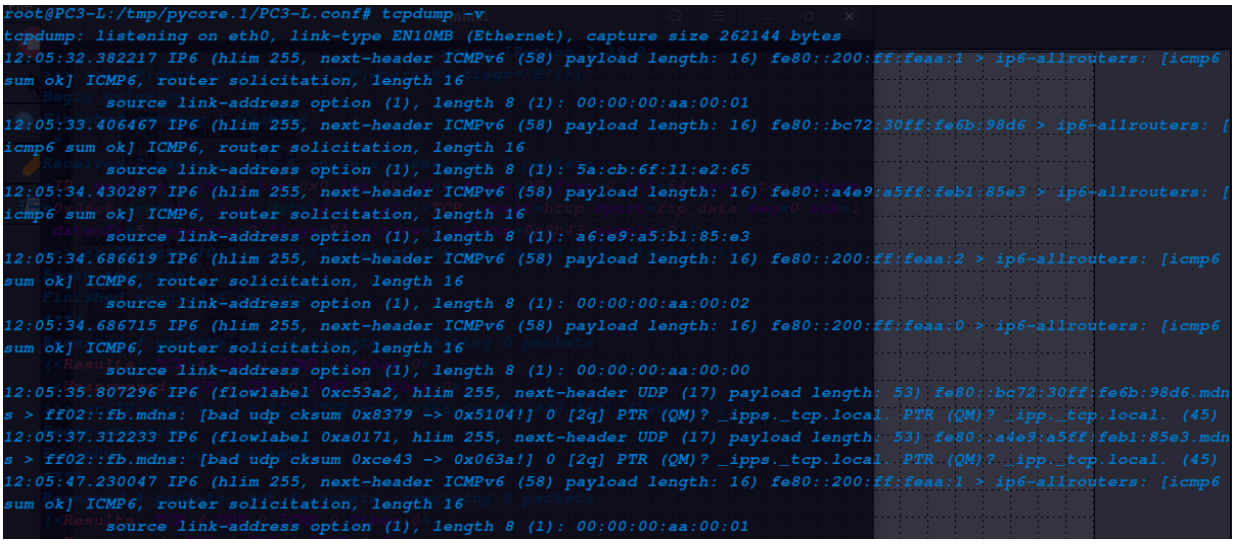

Figure 52 - Scapy Docs Usage Tutorial - SYN Scans - 2

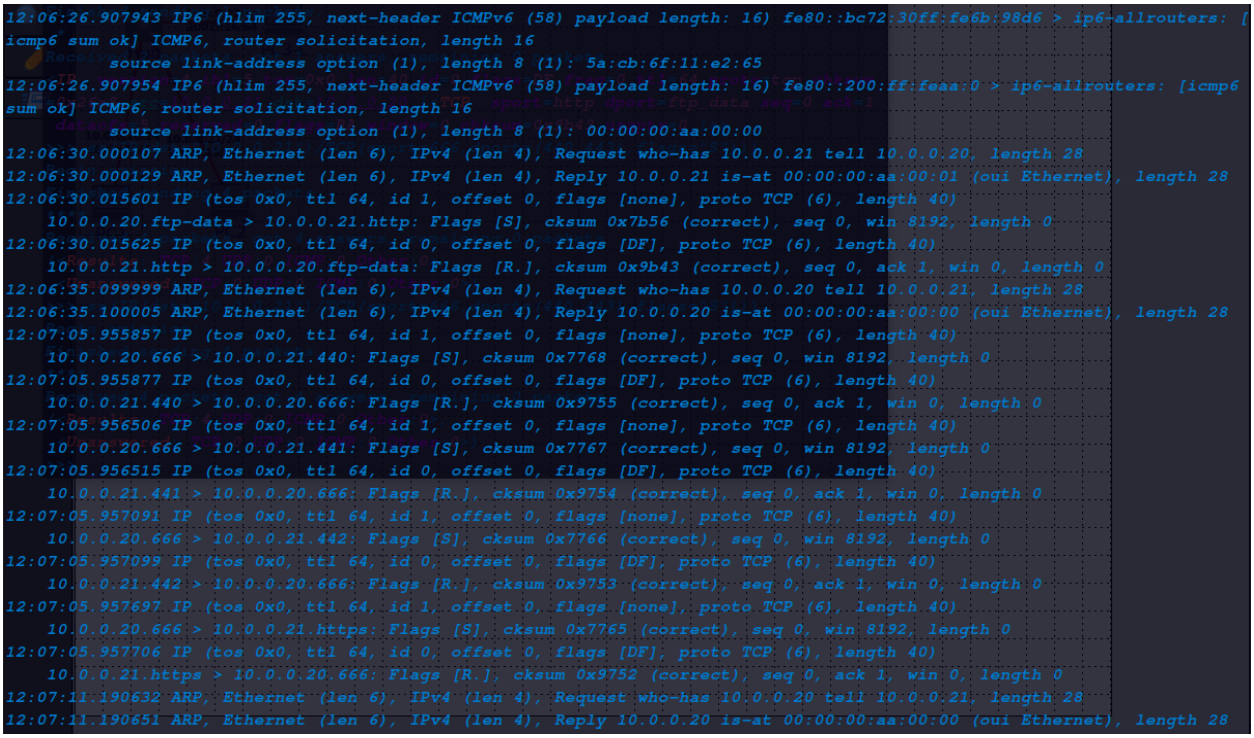

Figure 53 - Scapy Docs Usage Tutorial - SYN Scans - 3

- “ans, unans = _” & “ans.summary()” = (Figure 54)
- “ans.summary(lambda s,r: r.sprintf("%TCP.sport% \t %TCP.flags%"))” = (Figure 54).

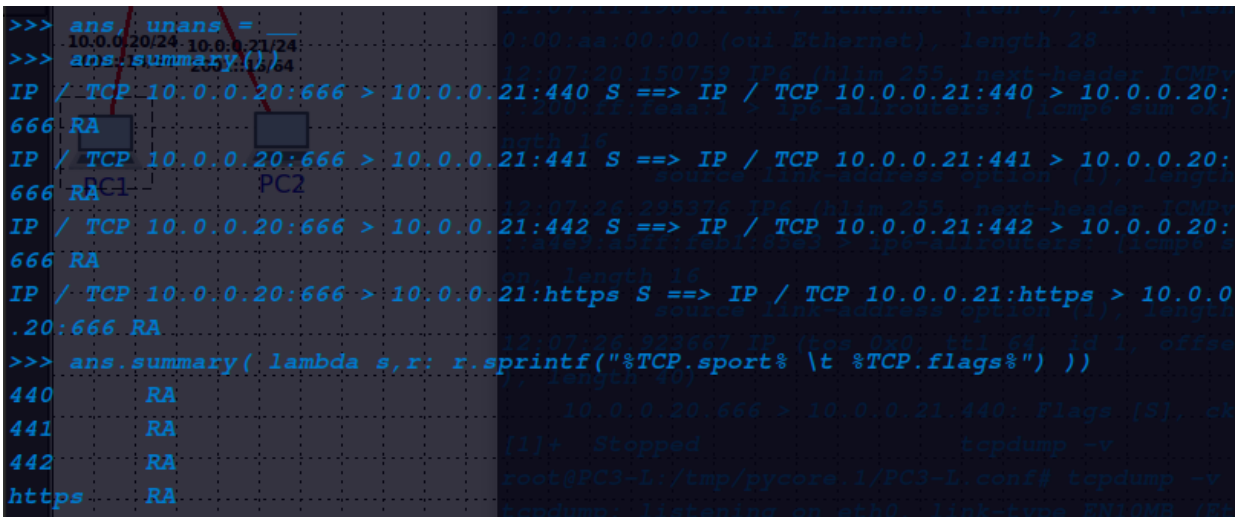

Figure 54 - Scapy Docs Usage Tutorial - SYN Scans - 4

TCPDump again was able to sniff traffic after the commands above executed. This captured again ICMP packets, ARP Ethernet who-has request/reply packets (Figure 55 & 56).

- "tcpdump -v"

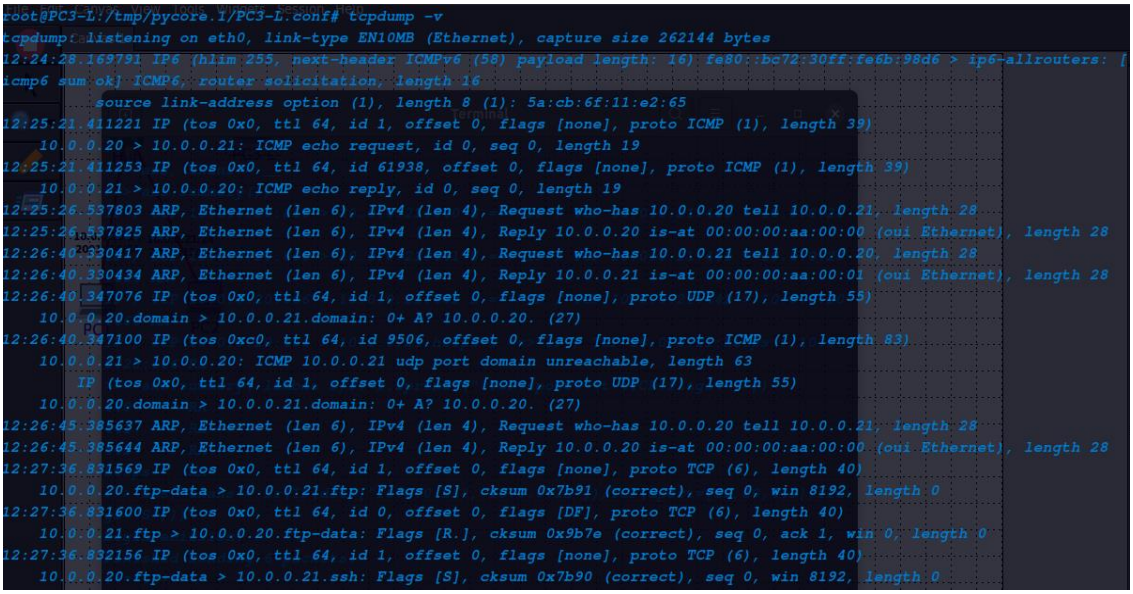

Figure 55 - Scapy Docs Usage Tutorial - SYN Scans - 5

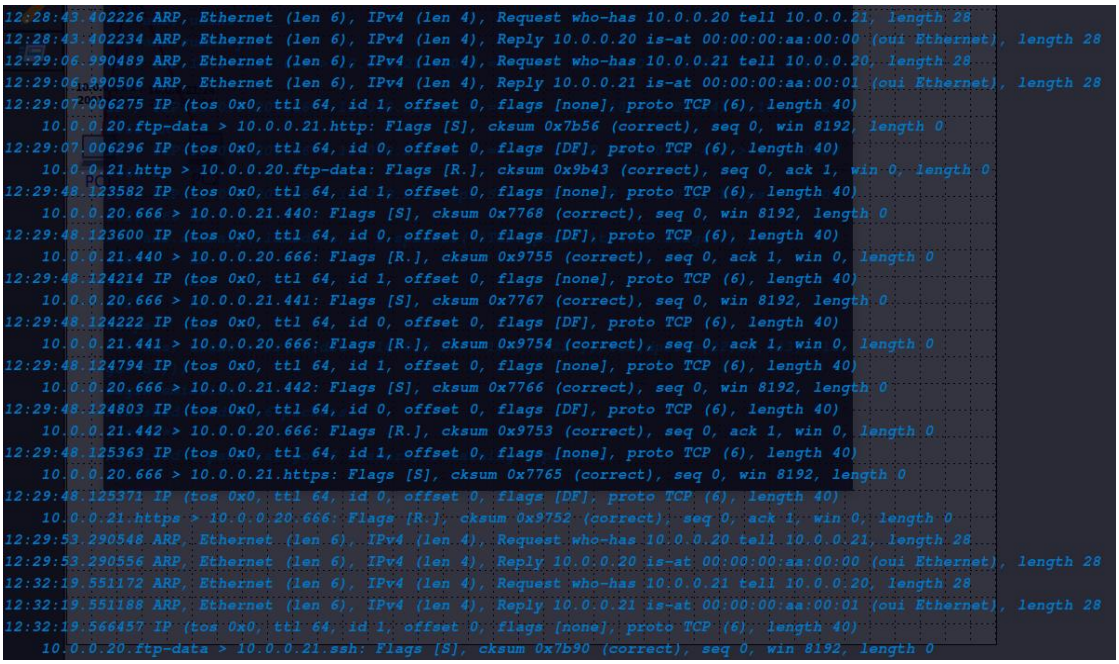

Figure 56 - Scapy Docs Usage Tutorial - SYN Scans – 6

- “ans, unans = sr(IP(dst=["10.0.0.21","10.0.0.22"])/TCP(dport=[22,80,443],flags="S"))” = (Figure 57).
- “ans.make_table(lambda s,r: (s.dst, s.dport, r.sprintf("{TCP:%TCP.flags%}{ICMP:%IP.src% - %ICMP.type%}"))) = (Figure 57).
- “ans.nsummary(lfilter = lambda s,r: r.sprintf("%TCP.flags%") == "SA") = No Output (Figure 57).
- “ans.summary(lfilter = lambda s,r: r.sprintf("%TCP.flags%") == "SA",prn=lambda s,r: r.sprintf("%TCP.sport% is open")) = No Output (Figure 57).
- “ans.filter(lambda s,r: TCP in r and r[TCP].flags&2).make_table(lambda s,r: (s.dst, s.dport, "X")) = No Output (Figure 57).
- “report_ports("192.168.1.1",(440,443)) = (Figure 57).

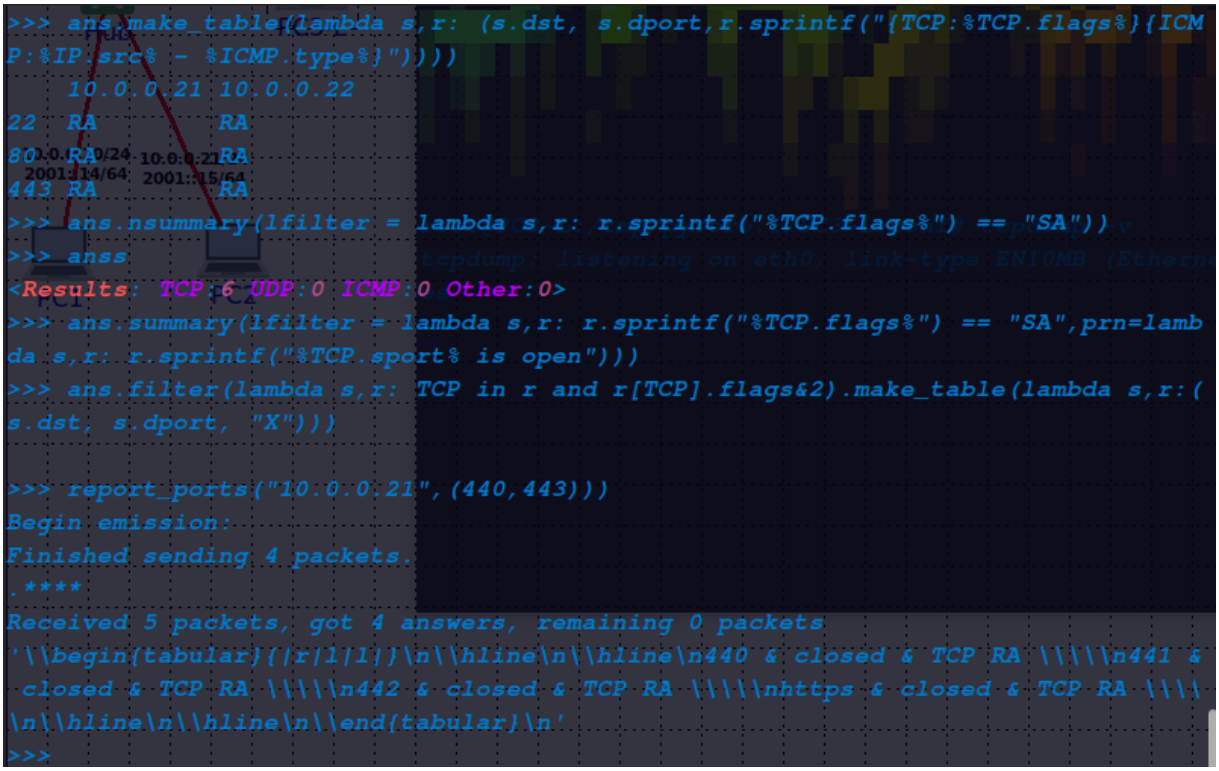

Figure 57 - Scapy Docs Usage Tutorial - SYN Scans - 10

TCP Traceroute

A classic Scapy TCP Traceroute already has “some other high-level functions are already coded” into it (Figure 58, 61 & 62). Scapy can “also use the GeoIP2 module, in combination with matplotlib and cartopy to generate fancy graphics” (Scapy, 2022). Figure 62 uses ‘the traceroute_map() function to print the graphic” however this did not print anything (Scapy, 2022). This requires that the “geoip2 module, its database” and “also the cartopy module” (Scapy, 2022).

- “ans, unans = sr(IP(dst=target, ttl=(4,25),id=RandShort())/TCP(flags=0x2))” = (Figure 58).

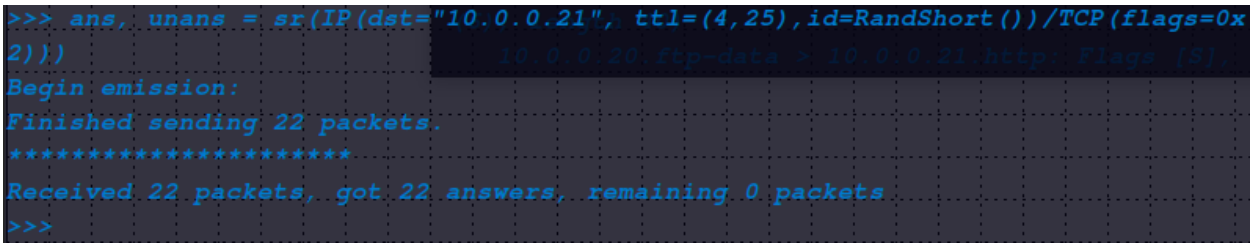

Figure 58 - Scapy Docs Usage Tutorial - TCP Traceroute - 1

TCPDump running on PC3-L was able to sniff some traffic from the previous commands. This is again very similar to previous results as there is ICMP packets, payloads, who-has/is-at replays and requests (Figure 59 & 60).

- “tcpdump -v”

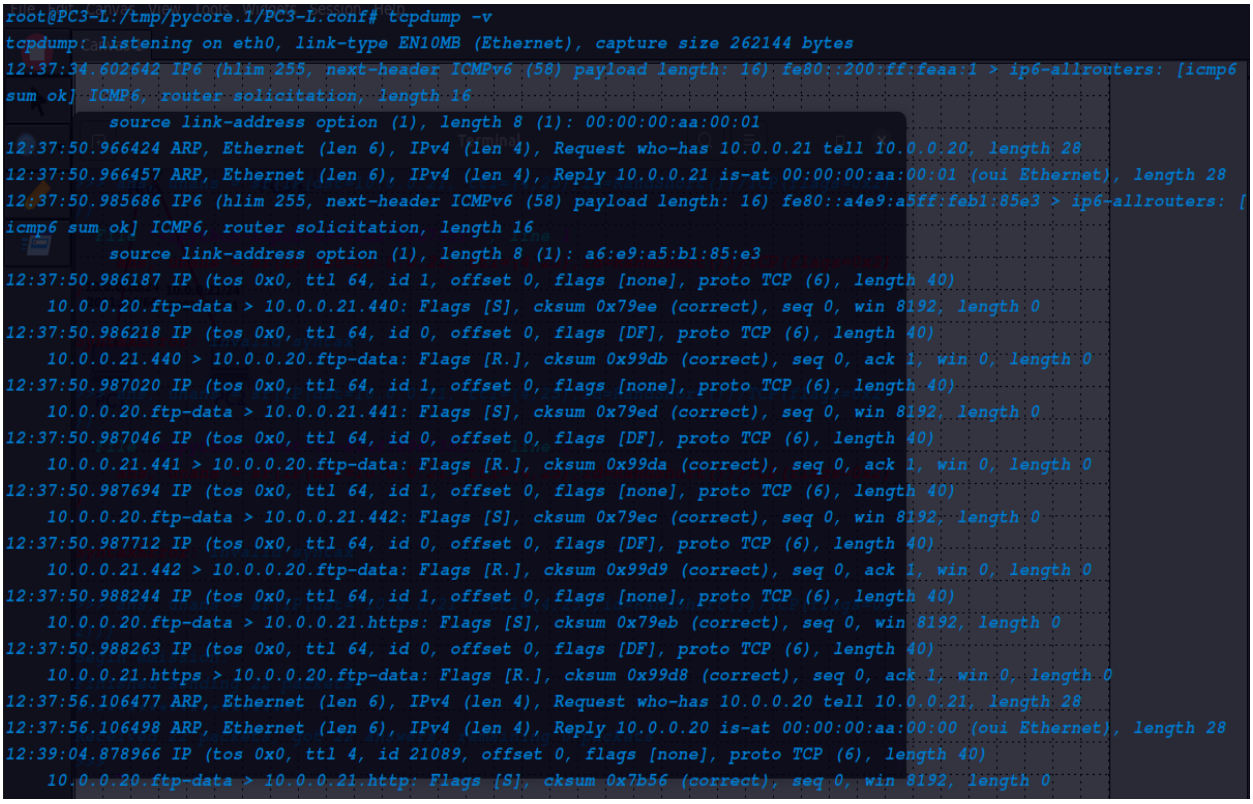

Figure 59 - Scapy Docs Usage Tutorial - TCP Traceroute - 2

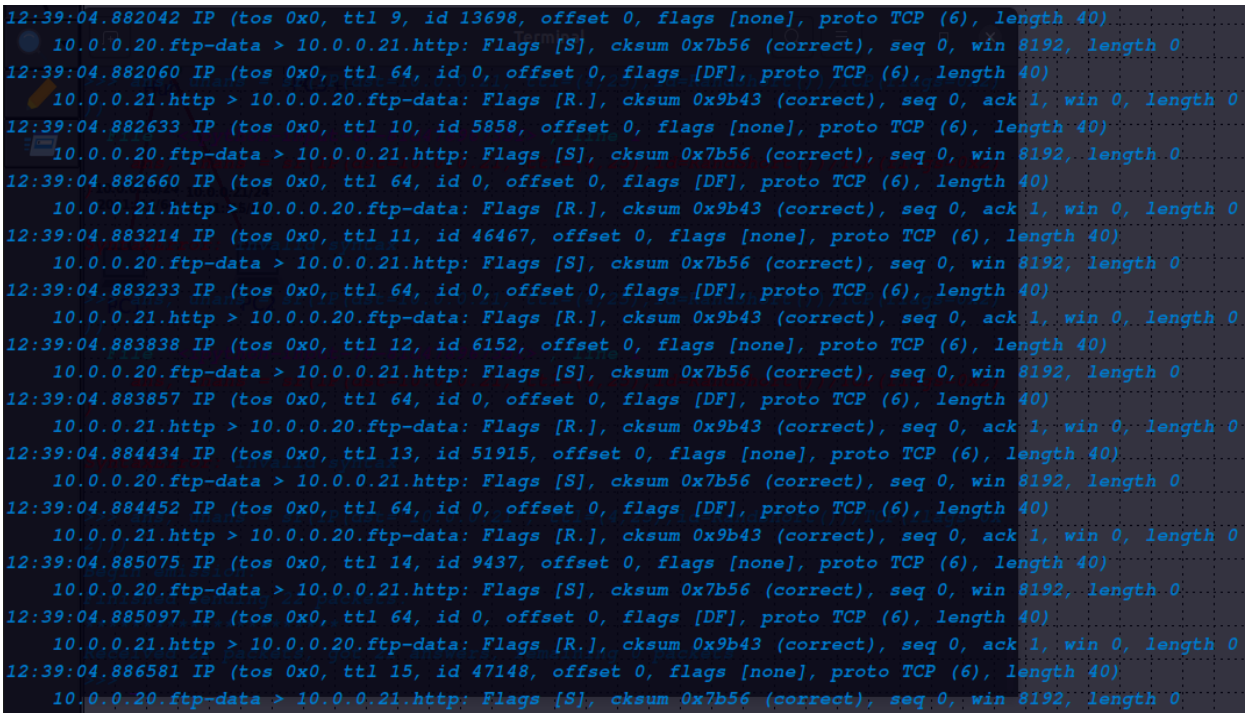

Figure 60 - Scapy Docs Usage Tutorial - TCP Traceroute – 3

- “for snd,rcv in ans: print snd.ttl, rcv.src, isinstance(rcv.payload, TCP)”

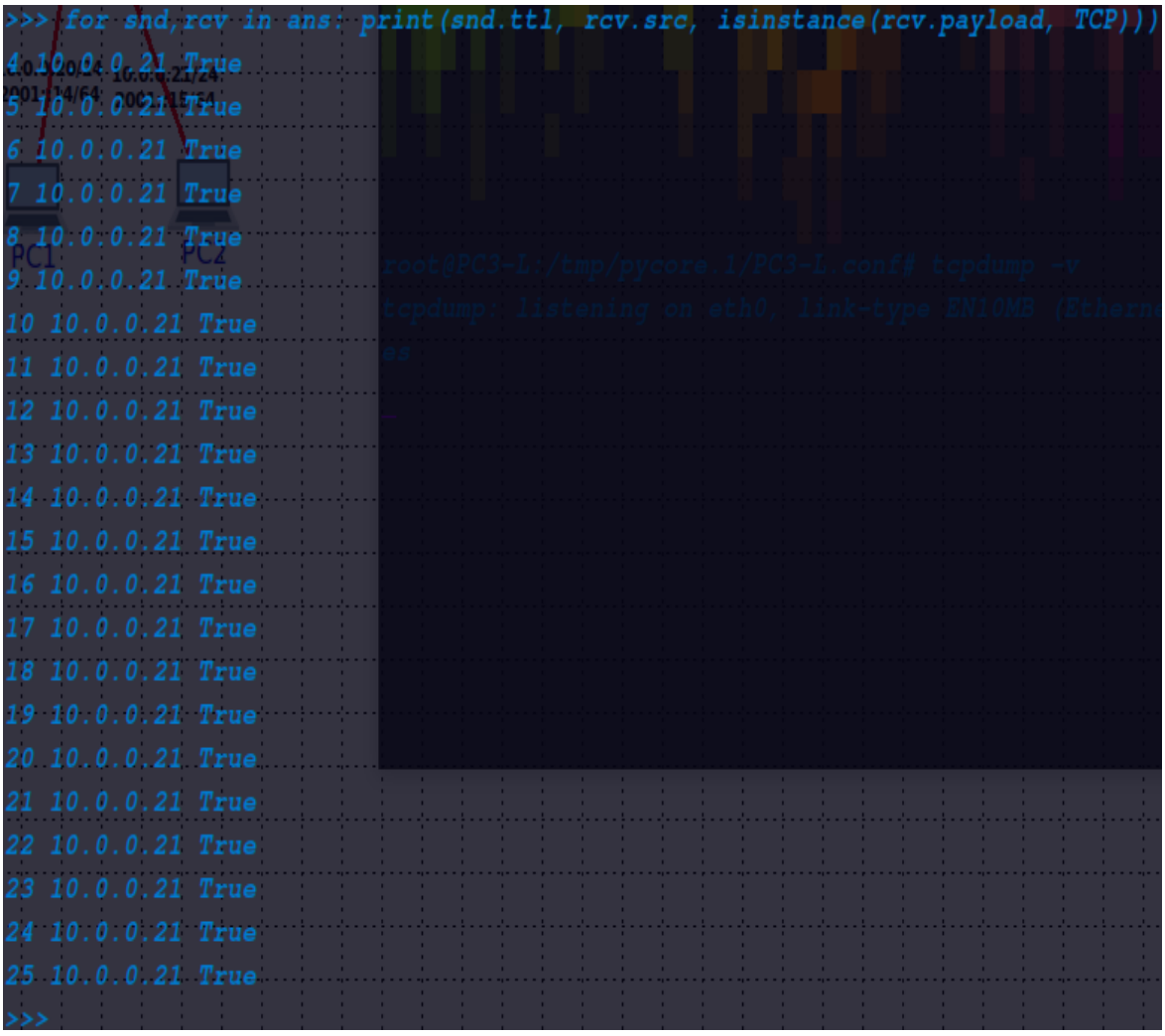

Figure 61 - Scapy Docs Usage Tutorial - TCP Traceroute – 4

- “lsc()” = (Figure 62).
- “conf.geoip_city = "path/to/GeoLite2-City.mmdb"” ← Why command didn’t work, no specified path… = (Figure 62).
- “a = traceroute(["www.google.co.uk", "www.secdev.org"], verbose=0)” = (Figure 62).
- “a.world_trace()” = (Figure 62).

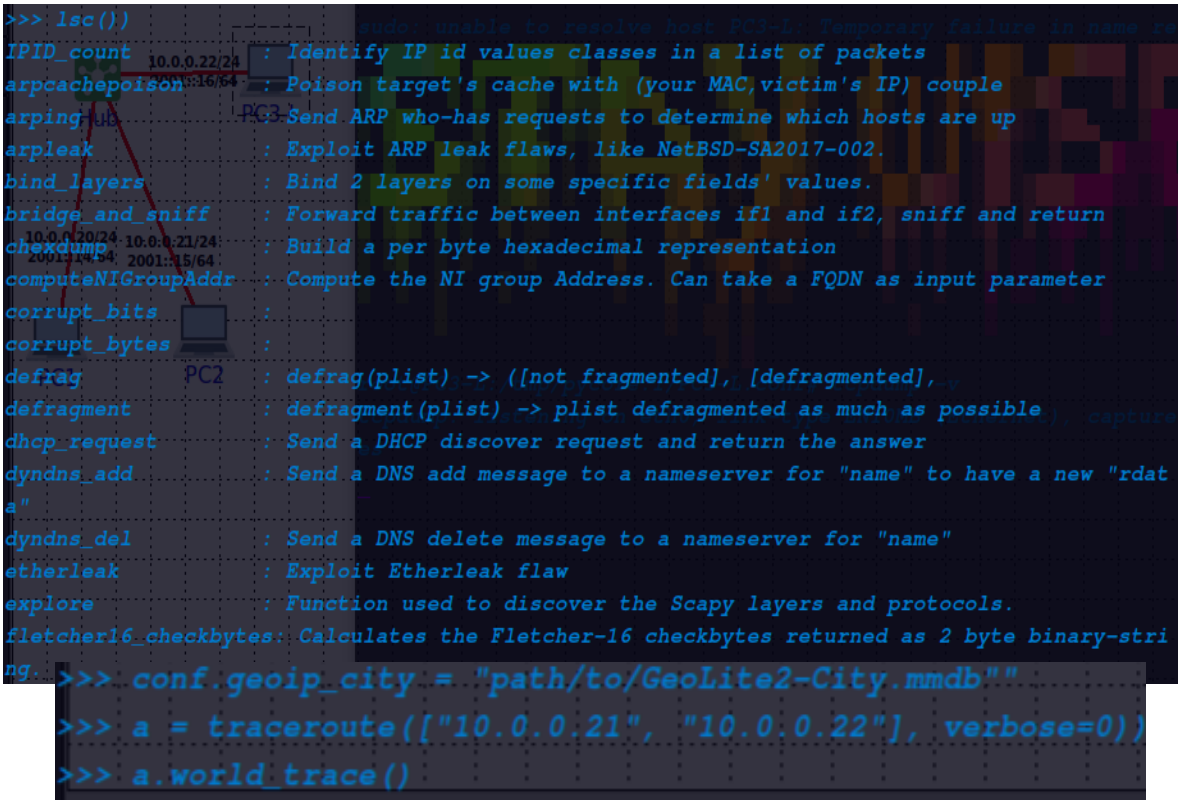

Figure 62 - Scapy Docs Usage Tutorial - TCP Traceroute – 5

Even though no graphic was generated because I forgot to specify a path and didn't notice for some reason, TCPDump actually picked up traffic. These again were the same as previous sections.

- "tcpdump -v"

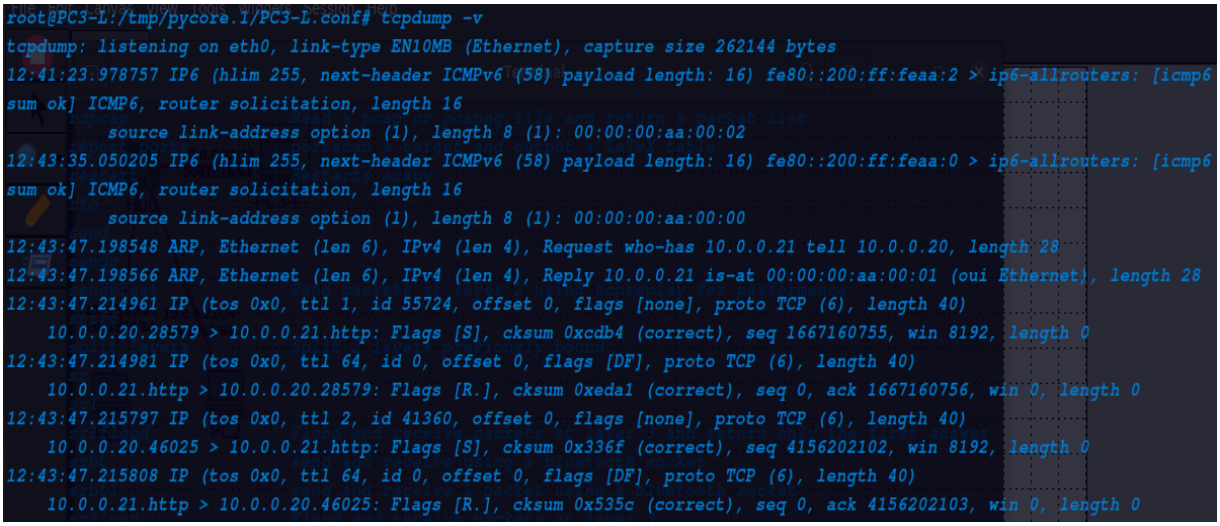

Figure 63 - Scapy Docs Usage Tutorial - TCP Traceroute - 6

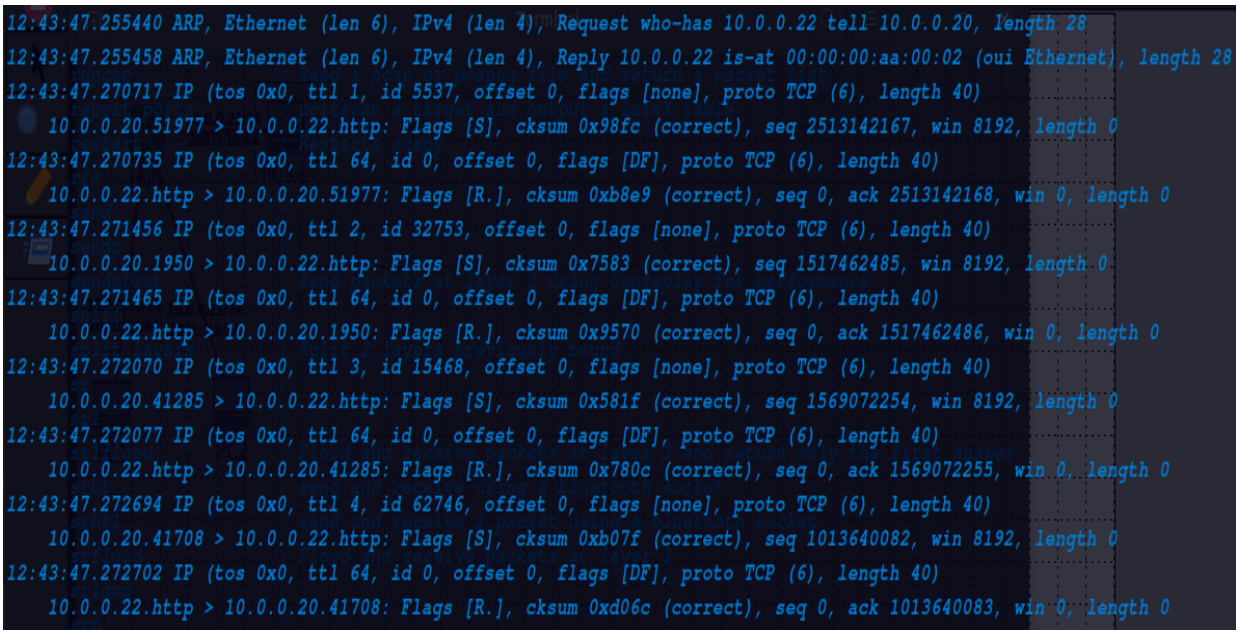

Figure 64 - Scapy Docs Usage Tutorial - TCP Traceroute - 7

Scapy Python Scripts

In this section, we will be taking scripts from the previous sections, and using them to create two different Python scripts that we can use to run on PC1 and target PC2, and then running TCPDump on PC3-L to attempt to sniff the traffic.

Setting-up .txt for Python Script

The first step is to create two .py files using the text editor. The first step is to open the text editor, select the little down arrow next to the words 'Plan Text', search for Python3 in the search bar and double click Python3 when it appears (Figure 65). This sets the language of the document to Python3. Then we can select the save option and name the document. These names must end in .py which makes the text files turn into the Python executable scripts needed by this lab. These were saved in the directory '/Desktop/432/py' (Figure 65). These are named "432py-1.py" & "432py-1.py (Figure 65).

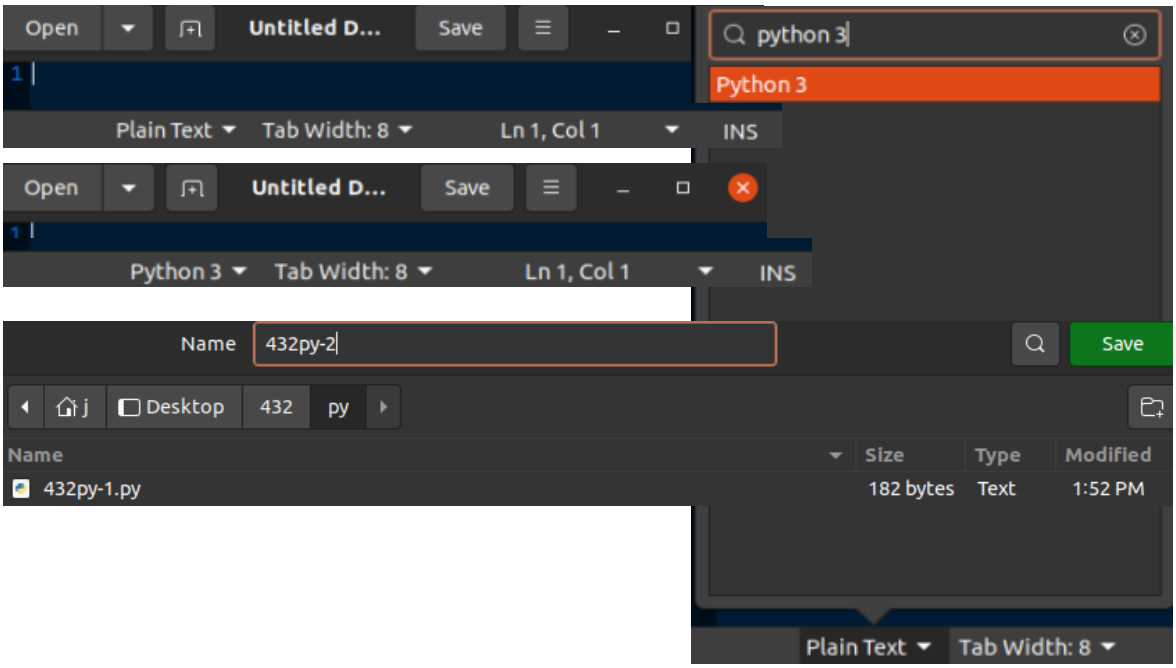

Figure 65 - Setting-up Text File

432py-1.py

The first script uses commands from the following sections, along with the sections, the command(s) used will be listed.

- Sending Packets:
    - “send(IP(dst="10.0.0.21")/ICMP())”
    - “sendp(Ether()/IP(dst="10.0.0.21",ttl=(1,4)), iface="eth0")”
    - “sendp(rdpcap("/home/j/scapy/test/pcaps/pfcp.pcap"))”
- Fuzzing:
    - “send(IP(dst="target")/fuzz(UDP()/NTP(version=4)),loop=1)”
- Injecting Bytes:
    - “pkt = IP(len=RawVal(b"NotAnInteger"), src="127.0.0.1")”
    - “print(bytes(pkt))”

432py-1.py

```
import scapy
from scapy.all import *

print("####################\nSending Packets\n ")
time.sleep(1)
send(IP(dst="10.0.0.21")/ICMP())
time.sleep(0.5)
print("----\n ")
sendp(Ether()/IP(dst="10.0.0.21",ttl=(1,4)), iface="eth0")
print("----\n ")
time.sleep(0.5)
sendp(rdpcap("/home/j/scapy/test/pcaps/pfcp.pcap"))
time.sleep(0.5)
print(" \n----\n \nDone.\n ")
time.sleep(0.5)

print("####################\nFuzzing - Use 'Ctrl + C' to Stop Script!\n ")
time.sleep(1)
send(IP(dst="10.0.0.21")/fuzz(UDP()/NTP(version=4)),loop=1)
time.sleep(0.5)
print(" \n----\n \nDone.\n ")
time.sleep(1)

print("####################\nInjecting Bytes\n ")
time.sleep(1)
pkt = IP(len=RawVal(b"NotAnInteger"), src="127.0.0.1")
time.sleep(0.5)
print("----\n ")
print(bytes(pkt))
time.sleep(0.5)
print(" \n----\n \nDone.\n ")
time.sleep(1)

```

Figure 66 - Python Script - 432py-1.py

432py-2.py

The second script uses commands from the following sections, along with the sections, the command(s) used will be listed.

- Send & Receive Packets: Pts. 1
    - “p = sr1(IP(dst="10.0.0.21")/ICMP()/"XXXXXXXXXXX")”
    - “print(p)”
    - “s = p.show(dump=True)”
    - “print(s)”
- Send & Receive Packets: Pts. 2
    - “sr1(IP(dst="10.0.0.21")/UDP()/DNS(rd=1,qd=DNSQR(qname="10.0.0.20")))”
    - “an, unan = sr1(IP(dst="10.0.0.21")/UDP()/DNS(rd=1,qd=DNSQR(qname="10.0.0.20")))”
    - “print(an, unan)”
    - “print(an.summary())”
- SYN Scan
    - “sr1(IP(dst="10.0.0.21")/TCP(dport=80,flags="S"))”
    - “ans, unans = sr(IP(dst="10.0.0.21")/TCP(sport=666,dport=(440,443),flags="S"))”
    - “print(ans, unans)”
    - “print(ans.summary())”
    - “print(ans.summary(lambda s,r: r.sprintf("%TCP.sport% \t %TCP.flags%")))”
    - “report_ports("10.0.0.21",(440,443))”

```python
import scapy
from scapy.all import *

print("####################\nSend & Revice Packet(s) - Pt.1\n ")
time.sleep(1)
p = sr1(IP(dst="10.0.0.21")/ICMP()/"XXXXXXXXXXX")
time.sleep(0.5)
print("----\n ")
print(p)
s = p.show(dump=True)
time.sleep(0.5)
print(" \n----\n ")
print(s)
time.sleep(1)
print("----\nDone.\n \n####################\nSend & Revice Packet(s) - Pt. 2\n ")
time.sleep(0.5)
sr1(IP(dst="10.0.0.21")/UDP()/DNS(rd=1,qd=DNSQR(qname="10.0.0.20")))
an, unan = sr(IP(dst="10.0.0.21")/TCP(dport=[21,22,23]))
time.sleep(0.5)
print(" \n----\n ")
print(an, unan)
time.sleep(0.5)
print(" \n----\n ")
print(an.summary())
time.sleep(1)
print(" \n----\n \nDone.\n \n####################\nSYN Scan\n ")
time.sleep(0.5)
sr1(IP(dst="10.0.0.21")/TCP(dport=80,flags="S"))
ans, unans = sr(IP(dst="10.0.0.21")/TCP(sport=666,dport=(440,443),flags="S"))
time.sleep(0.5)
print(" \n----\n ")
print(ans, unans)
time.sleep(0.5)
print(" \n----\n ")
print(ans.summary())
time.sleep(0.5)
print(" \n----\n ")
print(ans.summary( lambda s,r: r.sprintf("%TCP.sport% \t %TCP.flags%") ))
time.sleep(0.5)
print(" \n----\n ")
report_ports("10.0.0.21",(440,443))
time.sleep(0.5)
print(" \n----\n \nDone.\n ")

```


Figure 67 - Python Script - 432py-2.py

Running in CORE Network Topology

Now that we have created our two Python scripts using commands from the interactive tutorial on the Scapy Docs, we can now test these scripts out within the CORE network topology and attempt to sniff traffic using TCPDump. The set up to this is similar to other labs, as we will be communicating between nodes one and two (PC1 & PC2) and run TCPDump on node three (PC3-L). The first step we will take is starting TCPDump on PC3-L using the command "tcpdump -v" (Figure 68).

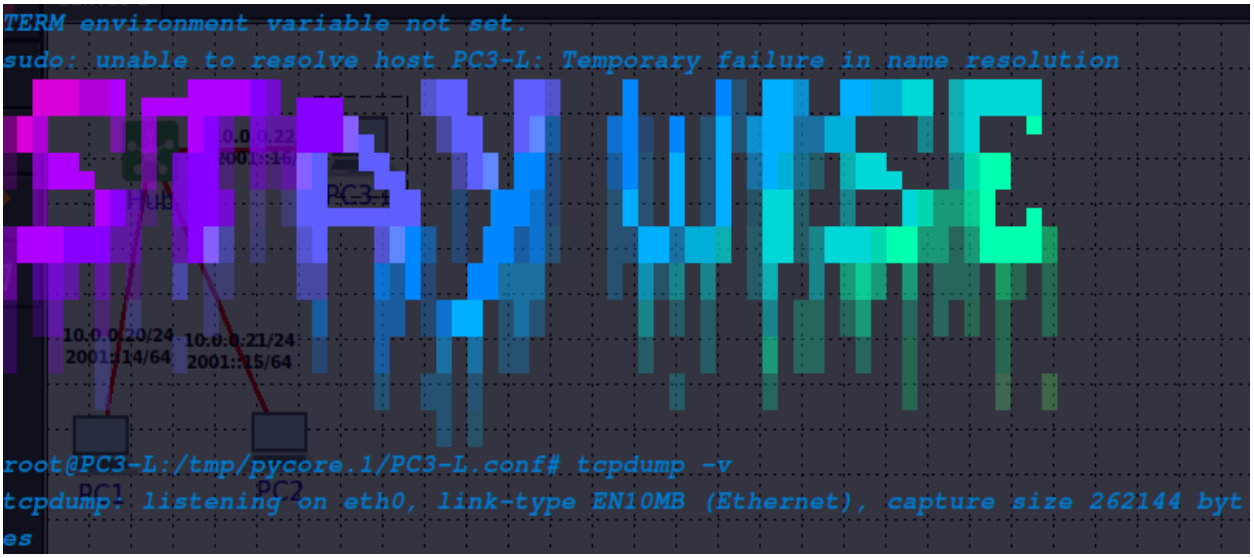

Figure 68 - Starting TCPDump in PC3-L

432py-1.py

Now that TCPDump is running on PC3-L, we can run the first Python script, 432py-1.py, by use of the PC1 terminal. After opening the terminal, I needed to change directories to the folder created previously 'py' which holds our Python scripts, then we are able to run the script (Figure 69 - 70). The output for this script is organized by section headers (sending packets, fuzzing, etc.), using lines like "print("----")" and blank lines like "print(" ")" (Figure 66). The output is also very similar to the outputs seen in the previous sections when using commands from the interactive tutorial. This is because they are the same commands and will provide the same output, this also applies for any traffic captured on PC3-L with TCPDump. The commands used to run the Python script are listed below.

- "cd /home/j/Desktop/432/py"
- "sudo python3 432py-1.py"

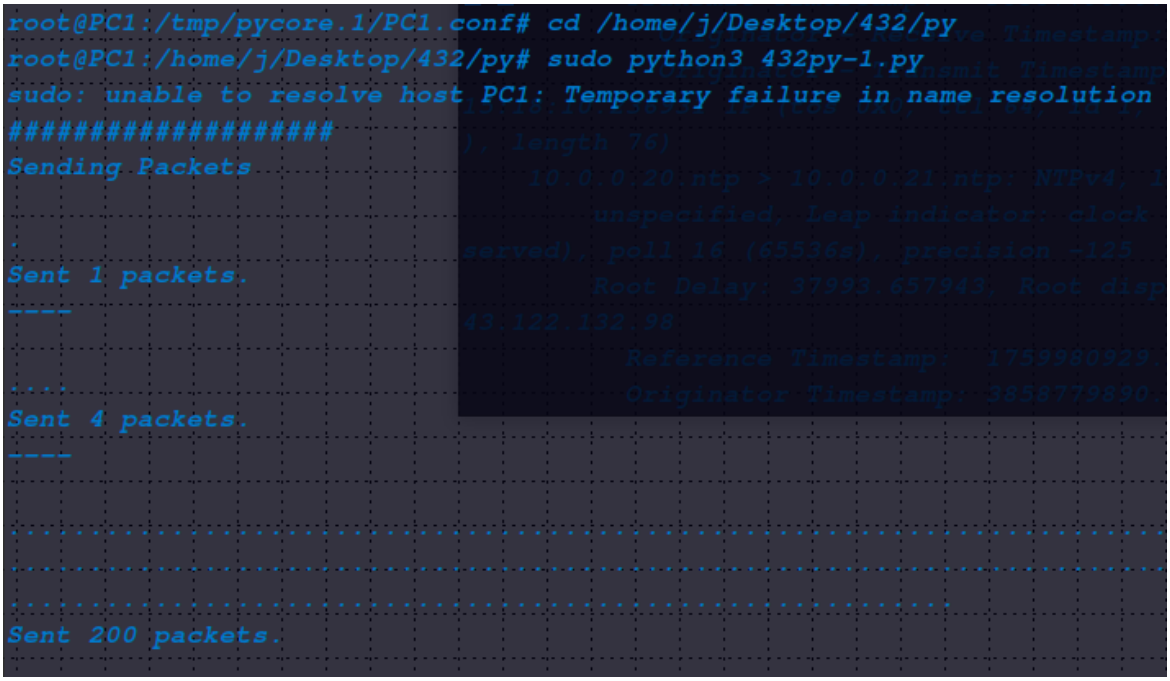

Figure 69 - Running 432py-1.py - 1

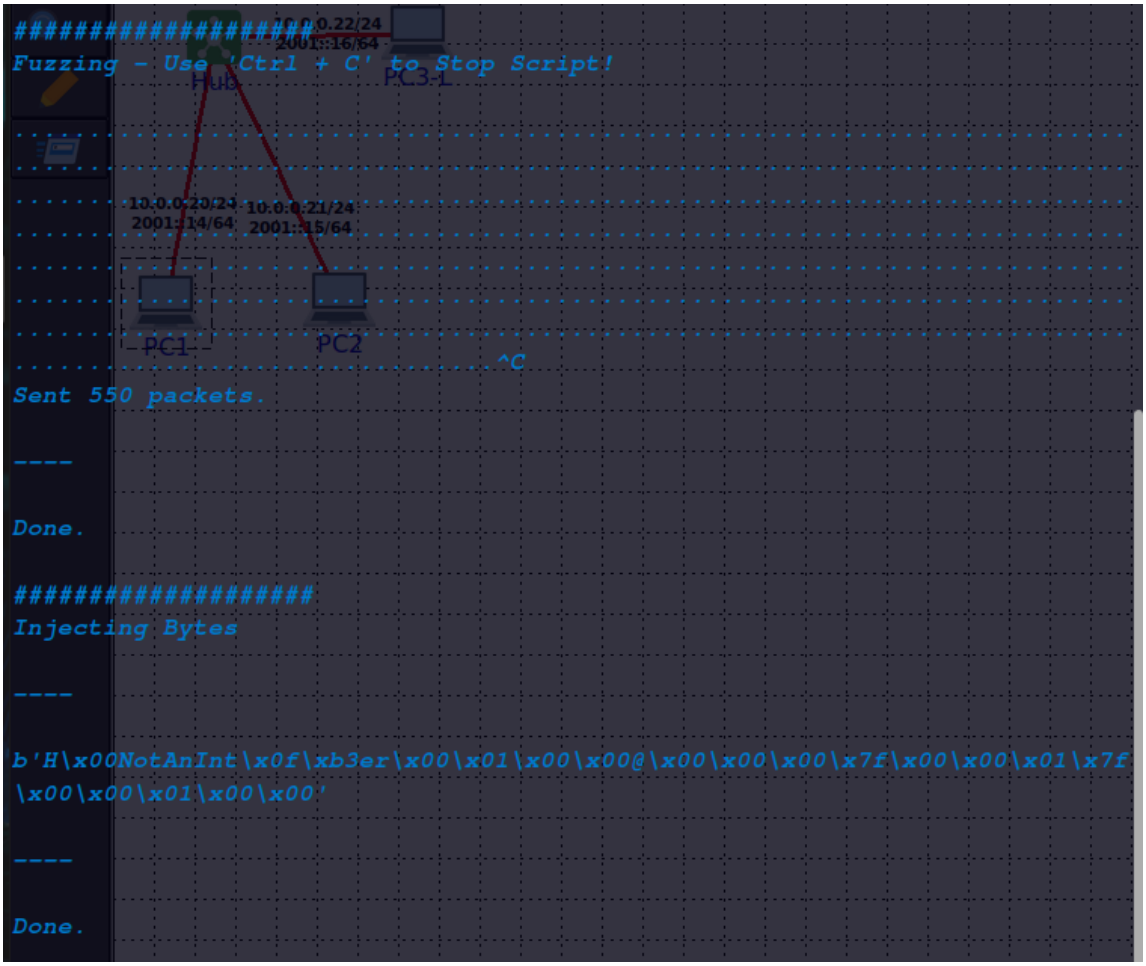

Figure 70 - Running 432py-1.py - 2

432py-2.py

After running the precious command, I closed the terminal for PC1 and re-opened it so I could run the second Python script we created, 432py-2.py (Figure 71 – 74). This second Python script has a longer output because it has more commands. The same rules from the prior script apply here. After the terminal was opened, I changed directories to the 'py' folder which holds our Python scripts, then I ran the script (Figure 69 - 70). The output for this script is organized by section headers (send & receive packets pt.1 & pt2, etc.), using lines like "print("----")" and

blank lines like “print(“ “)” (Figure 71 - 74). The output is also very similar to the outputs seen in the previous sections when using commands from the interactive tutorial. This is because they are the same commands and will provide the same output, this also applies for any traffic captured on PC3-L with TCPDump. The commands used to run the Python script are listed below.

- “cd /home/j/Desktop/432/py”
- “sudo python3 432py-2.py”

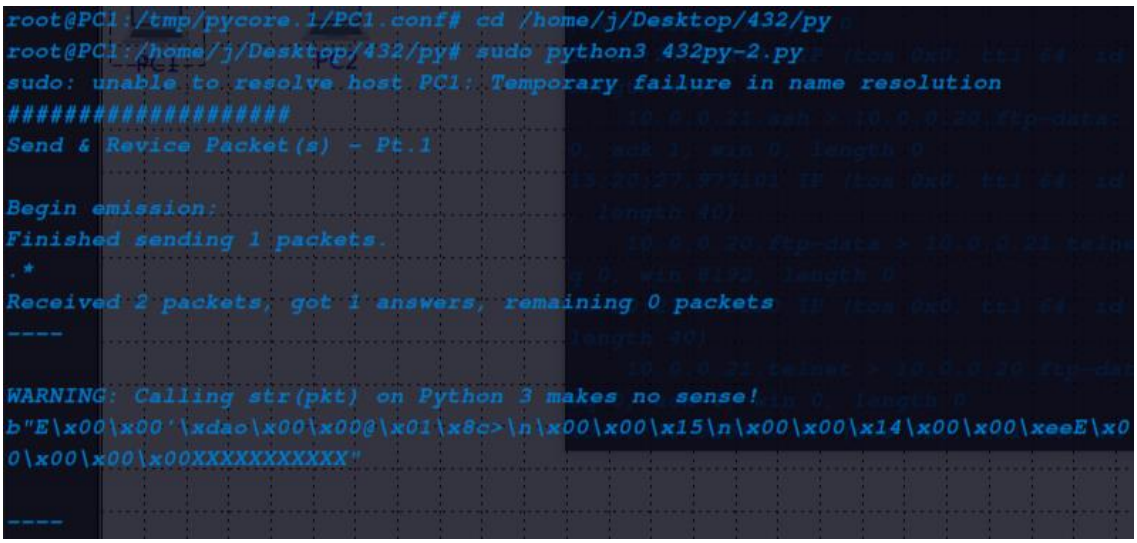

Figure 71 - Running 432py-2.py - 1

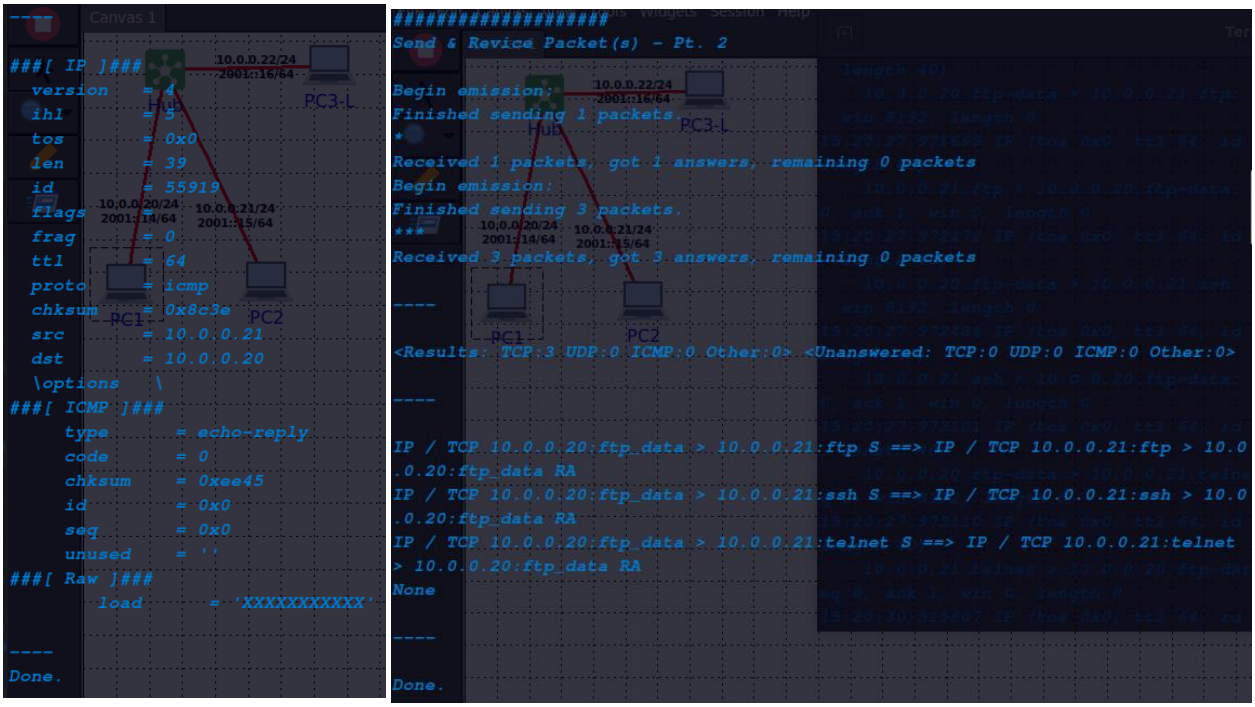

Figure 72 - Running 432py-2.py - 2

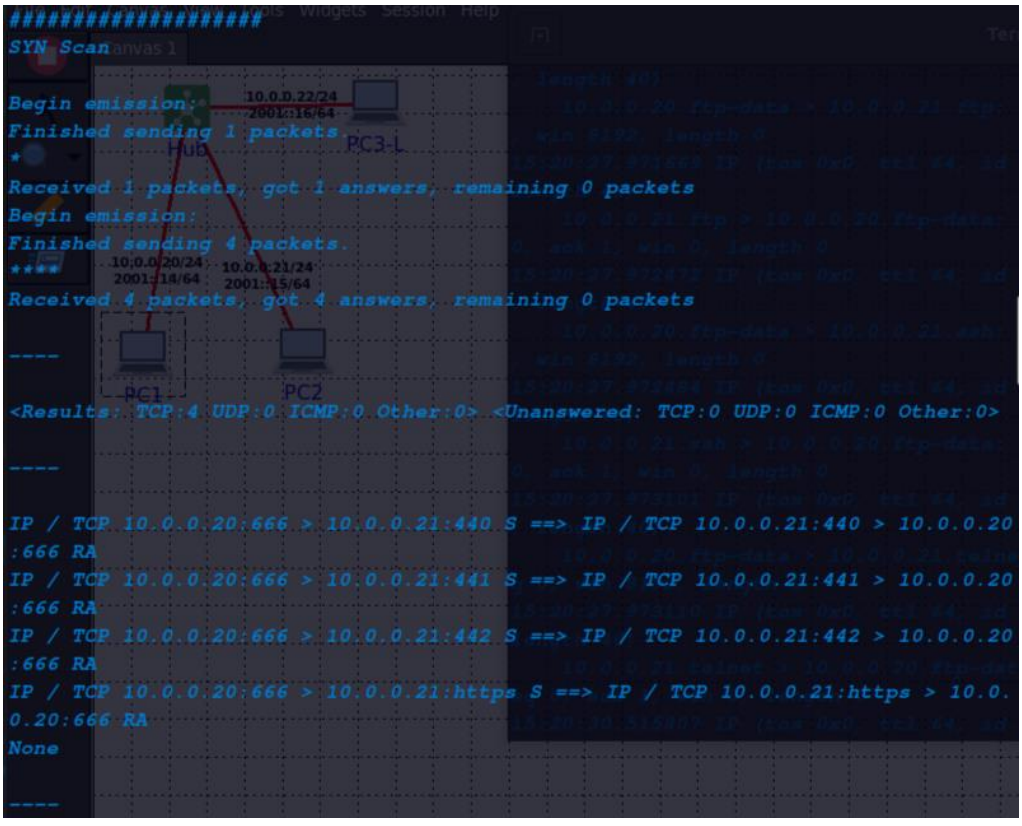

Figure 73 - Running 432py-2.py - 3

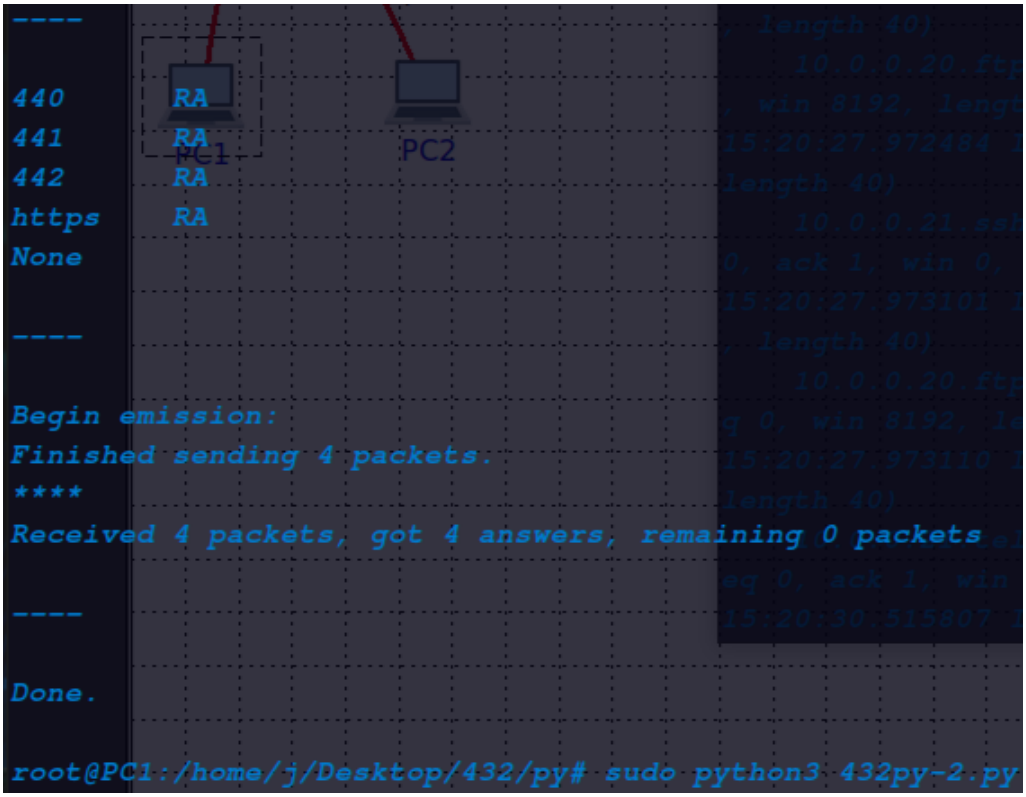

Figure 74 - Running 432py-2.py - 4

Scapy Results

After running each script, I gathered the TCPDump capture by taking screenshots and reviewed the data. This section will be the data which was collected. Above each Figure will be a description of what it contains.

432py-1.py

Let's first look at the traffic sniffed when running the first Python script, 432py-1.py (Figures 75 -77). After running the command "tcpdumo -v" in the terminal of PC3-L, and then running the first script, we can see that some data has been gathered (Figure 75). This mainly consists of the sending and receiving IP Addresses, ARP Ethernet who-has requests and is-at reply packets. There is also ICMP and ICMP echo reply packets. All of these were seen earlier in the interactive tutorial section.

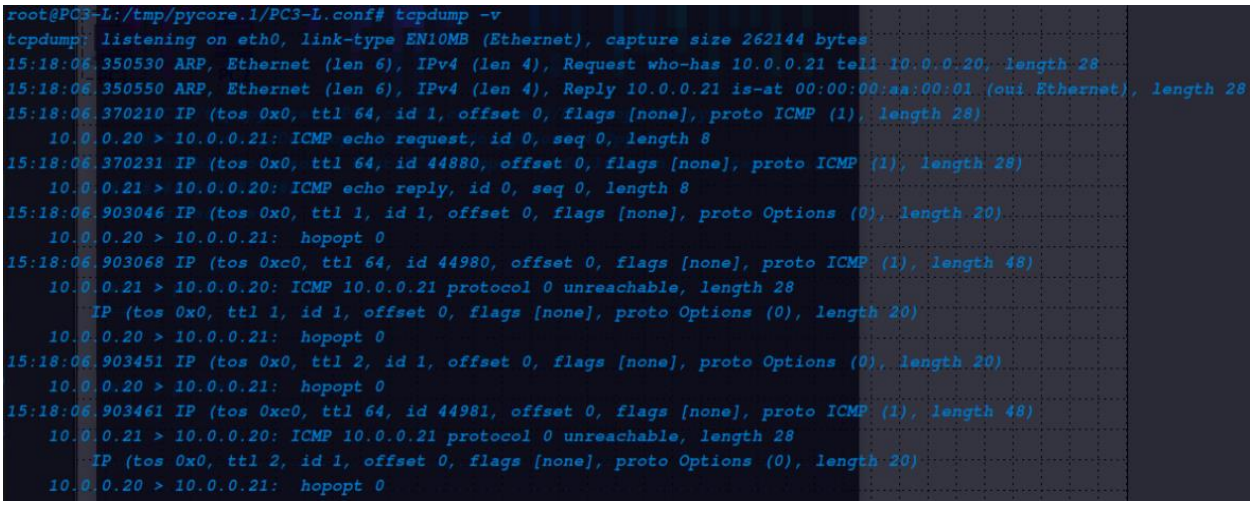

Figure 75 - Results from 432py-1.py - 1

After scrolling down, I saw again, unknown packets which were sending and receiving data. This consists of offsets, what looks like the headers to a file, then the plain text view of the 'message' (Figure 76).

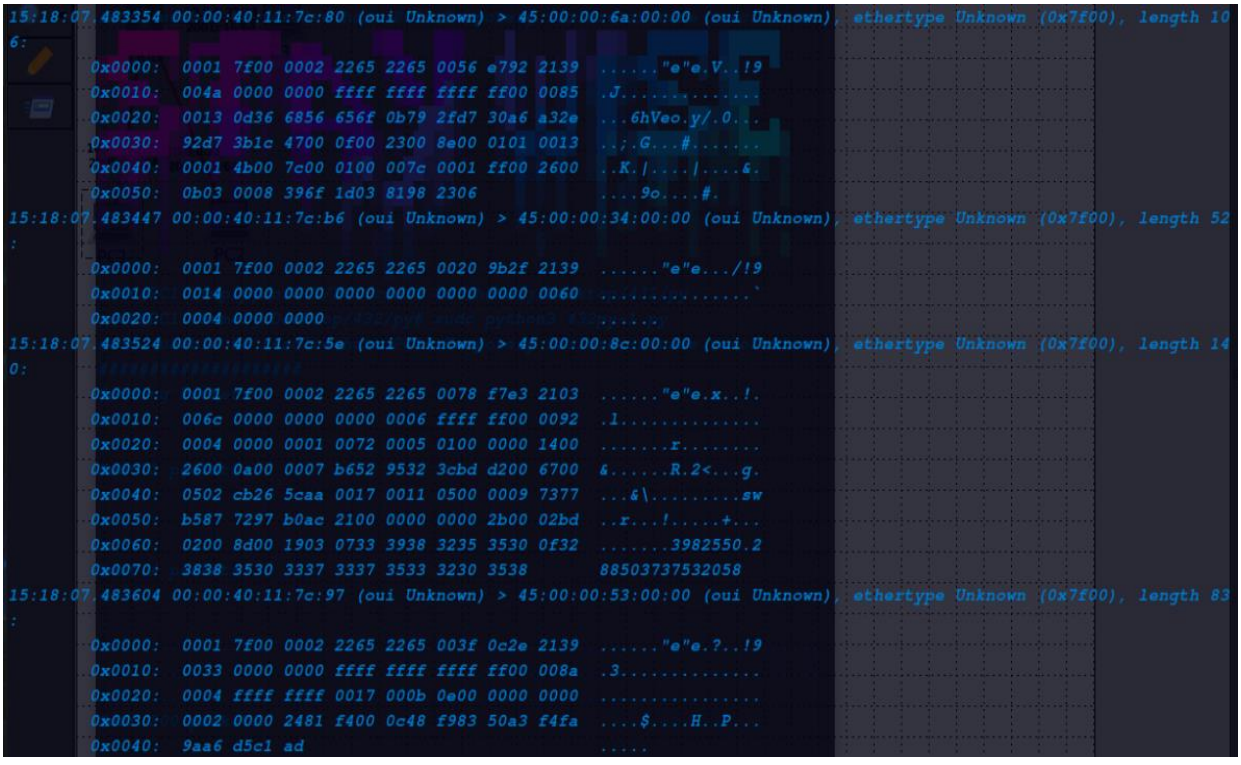

Figure 76 - Results from 432py-1.py - 2

Here is the NTPv4 packet capture again. This has a lot of valuable information which splits into two sections: Server and Root (Figure 77). The Server section contains: leap indicator, stratum, poll, and precision (Figure 77). The root section contains: delay, root dispersion, and reference ID (Figure 77). Below this, timestamp information is displayed.

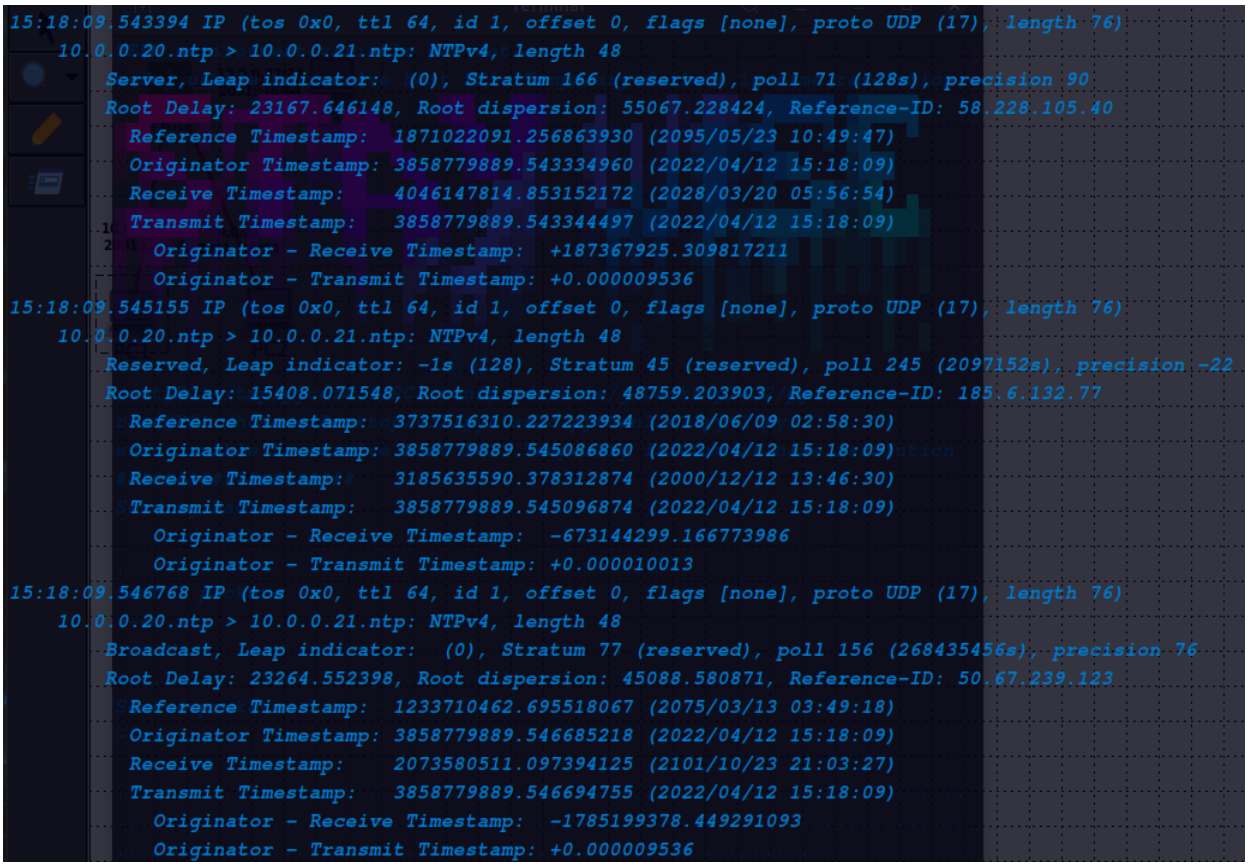

Figure 77 - Results from 432py-1.py - 3

432py-2.py

Now we can review the sniffed traffic after running our second Python script 432py-2.py (Figure 78 – 79). For the length of this Python script, the TCPDump capture surprisingly didn't sniff a lot of traffic. I preformed this on three occasions, the first was a test run. The second, I ran the script three times and still only this amount of data was collected, which I found odd.This data output is exactly the same as the interactive tutorial section. It contains the ARP Ethernet who-has request and is-at reply packets, ICMP and ICMP echo request packets, and some packets have FTP and checksum data (Figures 78 -79).

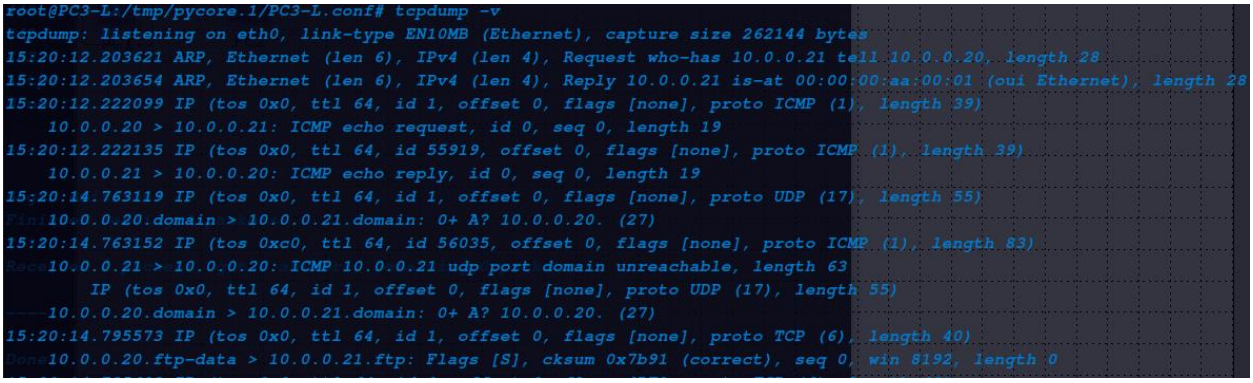

Figure 78 - Results from 432py-1.py - 4

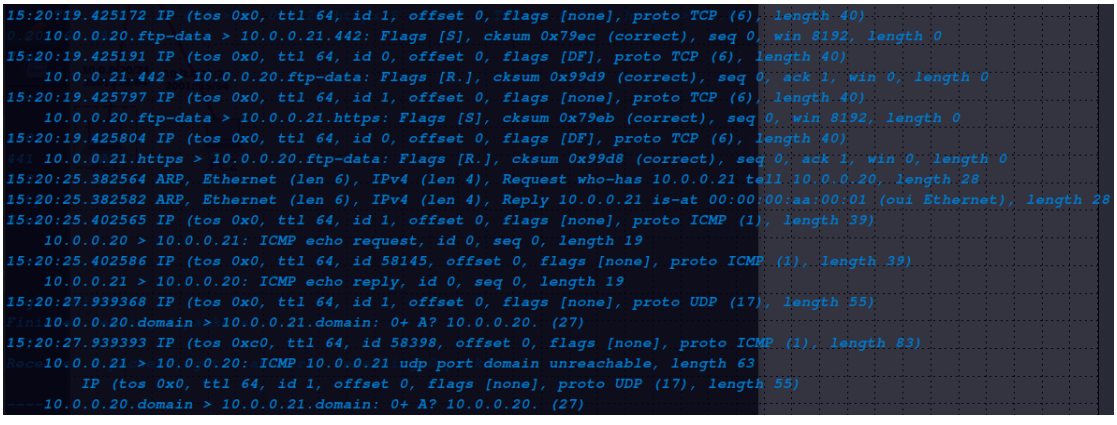

Figure 79 - Results from 432py-1.py - 5

Wrap-Up

After completing this lab, and closing CORE, I needed to provide a zip file which contains the two Python scripts we created. So do this, I first used the chmod command to change the privileges of the directory '/home/j/' as well all the recursive directories and their contents (Figure 80). I then changed to the Desktop directory and used the zip command to compress the directory '/432/' as well using the recursive flag, '-r', to automatically compress the contents of the other directories within. In this case, the '/py/' directory which contains are two Python scripts 432py-1.py and 432py-2.py (Figure 80). I then again used the chmod again to change the privilege level of the Desktop directory and opened the File Manager and directed to the Desktop folder (Figure 80). This is where the newly created zip file '432.zip' resides, and in doing so, I am able to drag and drop the .zip file from my Ubuntu VM to my host PC for turn in (Figure 80).

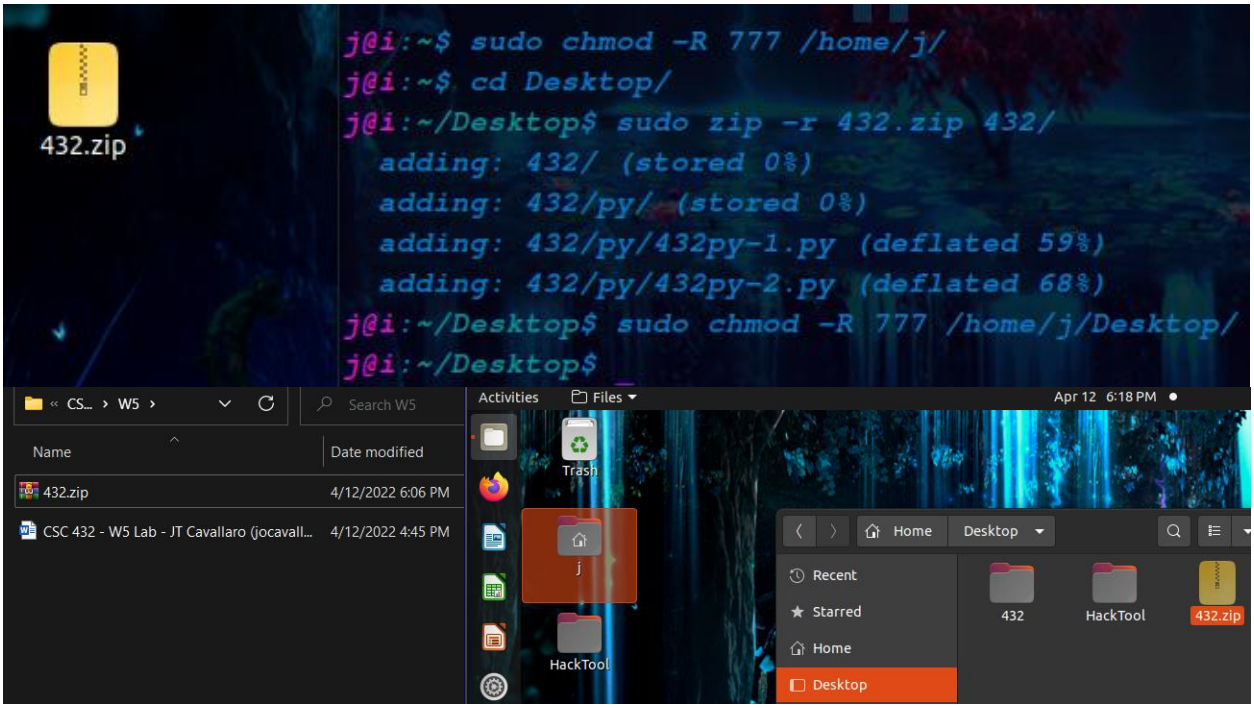

Figure 80 - Preparing Python Scripts for Turn-in

Conclusion

In the end of this lab, I was able to read through the Python3 Scapy Library Documentation, install Scapy and the Python3 Scapy Library, as well all of the needed dependencies, successfully. Then tested scapy out in the terminal of my Ubuntu VM to verify that everything was working properly. After building a network topology in CORE using three nodes and a hub, I ran the network topology and continued through examples in the docs. After reviewing the results, I created two Python scripts to be ran on the network topology. Using TCPDump, I was able to capture packets created and sent by the scripts and reviewed the results.

References

CORE, . (2022). *Using the core gui*. core. Retrieved March 20, 2022, from https://coreemu.github.io/core/gui.html & http://coreemu.github.io/core/install.html & https://github.com/coreemu/core

Scapy, . (2022). *Welcome to Scapy's documentation!* Welcome to Scapy's documentation! - Scapy 2.4.5. documentation. Retrieved April 10, 2022, from https://scapy.readthedocs.io/en/latest/ & https://scapy.readthedocs.io/en/latest/usage.html & https://scapy.readthedocs.io/en/latest/installation.html#installing-scapy-v2-x

TCPDump, . (2012). *Ubuntu Manpage: Tcpdump*. Ubuntu Manpage: Tcpdump - dump traffic on a network. Retrieved April 5, 2022, from http://manpages.ubuntu.com/manpages/trusty/man8/tcpdump.8.html

Appendix A: Tools –

Tool	**Version**
Google Chrome	99.0.4844.51
Engage	N/A
VMWare Horizon Client	2111
Ubuntu LTS Linux VM	20.04
CORE	8.2.0
Python2 & Python3	2.7.18 & 3.8.10
Scapy	2.4.5rc1.dev228

Appendix B: Host & Virtual Machine Specifications –
Host PC:

- Computer Name: PCJ
- Operating System (OS) Name: Microsoft Windows 11 Home
- OS Version: 10.0.22000 Build 22000
- System Make/Model: Dell Inc. G7 7588 x64
- System Serial Number: N/A
- Time Zone of Examiner Machine: Eastern

Ubuntu VM:

- Computer Name: ubuntu (Before Lab Completion)
- Computer Name: i2 (After Lab Completion)
- Operating System (OS) Name & Version: Ubuntu 20.04 LTS x64
- GNOME Version: 3.36.8
- Kernel: 5.4.0-104-generic
- LSB Version: core-11.1.0ubuntu2-noarch:security-11.1.0ubuntu2-noarch
- Time Zone of Examiner Machine: Eastern