Introduction

For this project we were tasked with sending covert communications to and from a machine protected by and intrusion detection system (IDS). IDS are used in many ways, from being able to stop large flows of traffic from a single machine, to being able to drop packets containing specific headers that look malicious. This project was broken down into four separate phases, the reconnaissance, the Infrastructure building phase, the attack phase and the defence phase. During the reconnaissance phase our group did a preliminary exploration on three IDS tools, packet inspection, flow monitoring, and SDN-monitoring, and how they are used while outlining the strengths and weaknesses of each. For the next phase we used our knowledge of Snort gained in the previous phase, along with experience from the last two missions, to set up an HTTPS server on an infected host. We were also able to have one of our virtual machine hosting Snort to monitor traffic, a client that would make normal HTTPS requests, and a command and control (C&C) VM. For our attack we decided to send messages between the C&C and infected web server using a covert channel hiding instructions inside an unexpected part of the TCP packet. To create cover “noise”/traffic for our covert channel packets, we created a Python script that would blast the network with HTTP traffic. We then defended against this by writing specific rules for Snort that would drop the packets if it detected if certain parts of the packet seemed to be malicious in a specific way. We were also able to write other Snort rules to be able to protect against the flood attack we created.

Reconnaissance Phase

Packet Inspection Tools

There are several packages available to automate the process of intrusion detection using packet inspection. Snort is one of these tools that is highly utilized and when implemented correctly can be used to great effect. Using metrics such as response times, packet types and packet contents, packet inspection tools can be configured to filter out specific types of traffic. Packet inspection and filters that can be used with it has several positive impacts on network traffic. For example, analysis at the packet level can allow admins to gain an accurate understanding of factors slowing down the network. Using packet inspection, admins can also track relevant traffic across their network and gain a better picture of network performance. (11) This can enable them to quickly isolate and address issues before they cause end-user impacts. In addition packet filters can pre-empt possible similar types of issues and things that could affect the end-user. While Packet inspection tools can block many existing vulnerabilities it is still a vulnerable option, just providing extra layers of defense and control of the network. For example, they can be overburdened by high amounts of information and let malicious packets through. Furthermore, the relative strength of the packet inspection tool as a defensive measure is proportional to the strength of the admins configuration of it and how that matches up with the type of attack leveled against it. In other words, they are not infallible as well disguised malicious packets can be let through because the packet inspection tool simply did not identify them as an attack.

Flow Monitoring Tools

There are a number of tools that monitor the flow of packets within the network. Tools such as NetFlow, sFlow, and JFlow are a few commonly used tools to accomplish this task (2). These flow based monitoring tools, developed to work with clients such as Cisco, HP, and others, work by collecting packet information. It uses algorithms to be aware of who is creating traffic (where traffic is coming from), how much traffic is being generated, and where the traffic is being directed to (3). These tools lead to a number of positive impacts on the network. These include the productivity of the network - being able to have max performance and throughput for packets going over the network, understanding where bottlenecks on the network occur and give insights for possible solutions, and detections of security anomalies within the network (4). These flow monitoring tools, however, commonly lack accuracy and scalability. As the network traffic increases, the tools start to lose their ability to be as accurate as they would be when there is a lack of packets (5). An attacker that might want to infiltrate a network would want to slip packets between normal "noise" traffic (random GET, POST, DNS, etc) so that they do not stand out as an anomaly by these tools. Their goals are to be able to compromise the security goals of these devices, which include authenticity - the packets that are attempting to communicate with a given computer is really coming from who they claim to be - and accessibility - being able to ensure that traffic flow is not bottlenecked between links/routers. If they are able to successfully compromise these security goals, they could cause enough traffic to create a Denial of Service (DoS) attack or hop through the network, affecting computers without the tools being able to detect anything suspicious.

SDN Based Monitoring Tools

SDN, or software defined networking, allows a central software to holistically control the flow of packets in a network. SDN allows networks to more securely control the flow of packets in a network. This does include a high latency and larger packet overhead. A common protocol for SDN is OpenFlow which is a protocol that allows controllers to control the flow of packets in an SDN. In the research paper "Validating Security Protocols with Cloud-Based Middleboxes" by Taylor and Shue they use OpenFlow and routers that support OpenFlow to send packets needed for inspection to a middlebox in the cloud. Middleboxes are useful in SDN as they can inspect and monitor packets for specific network flows. Taylor and Shue used this method to keep latency down for many web packets, like for video games or internet connected televisions. SDN is very advantageous as it is an automated service that can handle selected packet flow.

Infrastructure Building Phase

For our Infrastructure, we had normal HTTPS servers, a botnet Command and Control machine and our IDS, Snort, running. The machine running Snort is on 10.4.7.3. This machine is also running Barnyard2 while will read the output of Snort and add it to a MySQL database that is also running on that machine. The machine that is running the Command and Control is on machine 10.4.7.5. This Command and Control opens a connection to the infected hosts on port 4443. We also want to generate legitimate traffic. We used the python script simple-https-server.py to start an https server and wait for connections. We generate this traffic by running real_traffic.py on VM 10.4.7.4. The https server uses a self signed certificate and private key to secure itself. We chose this over using a separate VM to run as a CA because it still generates the correct traffic and is sufficient for our purposes. That purpose mainly to show that the defense will not attempt to block legitimate traffic and it will not interfere with the attack.

IPTABLES and IP Routes:

```
$ 10.4.7.3
Ifconfig eth0:1 172.168.1.3 netmask 255.255.255.0 up
Ifconfig eht0:2 192.168.1.3
Iptables -A FORWARD -p tcp -m tcp -j NFQUEUE --queue-num 4 --queue-bypass
$ 10.4.7.4
Ifconfig eth0:1 172.168.1.4 netmask 255.255.255.0 up
Route add default gw 172.168.1.3
$ 10.4.7.6
Ifconfig eth0:1 172.168.1.6 netmask 255.255.255.0 up
Route add default gw 172.168.1.3
$ 10.4.7.2
Ifconfig eth0:1 192.168.1.2 netmask 255.255.255.0 up
Route add default gw 192.168.1.3
Iptables -A OUTPUT -p tcp -m tcp -j NFQUEUE --queue-num 5 --queue-bypass
$ 10.4.7.5
Ifconfig eth0:1 172.168.1.5 netmask 255.255.255.0 up
Route add default gw 178.168.1.3
Iptables -A OUTPUT -p tcp -m tcp -j NFQUEUE --queue-num 5 --queue-bypass
```

To test normal function of infrastructure do the following:

```
$ ssh cs4404@10.4.7.2
> sudo su
# python3 simple-https-server2.py
$ ssh cs4404@10.4.7.4
> sudo su
# python3 real_traffic.py
```

After running this, you should see something that resembles the outputs in figure 4 in appendix A.

Attack Phase

For our attack, we were able to successfully hide our instructions to the corrupted server in a covert channel hidden within the TCP header. To be able to change our TCP packet headers, we used Scapy and nfqueue and hid our commands in a spot of a packet that would not normally contain information. We were able to turn a specific request into integer values to be able to store as bytes in the Urgent Pointer field of the TCP, HTTPS packet. The urgent pointer field is simply a two byte field that can indicate if a packet needs to be sent immediately or not. It is transmitted over thirteen packets due to the fact that the size of the Urgent Pointer field is only two bytes. The command that we sent was " uname -r ". This would allow us to gather information from the operating system of the computer that we were attacking in order to represent the gathering of data from the machine. The infected server knows when the instruction starts and ends due to the first two packets and last two packets having the urgent pointer "32". After the machine receives the full instruction, the information is put into a file. When the attack_server.py program sees that file, it runs the contents of the file and then translates it into integers so it can be sent back to C&C in the same way the server first received the instructions, ie. the urgent pointer field. We set up a command-and-control that would be facilitating the attack. We are assuming that the attacker has access to both 10.4.7.6 (the machine that is able to blast the network with HTTPS traffic) as well as 10.4.7.5 (the machine that acts as the command-and-control machine). To run this attack, the C&C can blast the network with the HTTPS "noise" traffic and then send the packets to the infected machines that are running the server to listen for the instructions. We implement this with a multi-threaded Python script that blasts our channel with "noise". Originally the purpose of that was to flood snort with packets to sneak our attack through. It turns out that snort completely dropped all traffic when this happened, so it would not work in the way we desired. We still include it with a lower amount of sent packets, however, because it can still be useful to hide the attack from any human users watching, in real time or after the fact, because of the sheer volume of sent packets.

To test the attack you need to run several programs: (can also be run at the same time as the infrastructure without interference)

```
$ ssh cs4404@10.4.7.5
> sudo su
# python3 instruction_response.py
# queue_edit.py
$ ssh cs4404@10.4.7.2
> sudo su
# python queue_exit_resp.py
# python3 server_scapy_test.py
# python3 attack_server.py
# python3 simple-https-server2.py
$ ssh cs4404@10.4.7.4
> sudo su
# python3 real_traffic.py
$ ssh cs4404@10.4.7.6
> sudo su
# python3 gen_background_traffic.py
```

Lastly, in cs4404@10.4.7.5 run:

```
# python3 attack.py
```

After running this, you should see something that resembles figure 1 and 2, at the start and end respectively, in Appendix A.

Defense Phase

We made a multi-step defense that includes the packet monitor Snort. Using specific Snort rules along with SQL statements, we were able to drop all of the packets from the attempted flood of the network. Snort drops packets from an IP address if they send more than 70 packets within 10 seconds. We found this number to be appropriate as most TCP should that we were testing should not exceed this. We had to configure Snort to run NIDS mode. This required configuring Snort, DAQ, and barnyard2 to accept that mode. We had to download all NFQ for devs. We also had to create unique rules that satisfied our requirements. This took a lot of research and digging through snort's enormous manual and testing. Snort now looks for multiple things when dropping packets. First it looks for content in the payload that could contain messages that a botnet could use. This required the attack to use a covert channel. By exploiting the byte size in the urgent pointer the attack was able to bypass snort. We looked at the flags being sent by the attacker. The flags included the urgent flag which Snort looked at to determine if the packet needs to be dropped or not. Since the urgent pointer is used to be exploited by the attacker it drops the packet. Snort can be used in multiple ways to drop packets using other exploits. We can have Snort check the TTL or the TOS for specific numbers or bytes. This allowed us to block packets that we knew had specific exploits. We also have Snort using Barnyard2 to write to a mysql database. This allows the user to have database queries to see what traffic has been dropped, alerted, or allowed through.

We had more attempts to do more with the defense but because of bugs within certain libraries this became more difficult to do. There was a bug with NFQUEUE that did not allow multiple queues to have the same packet in a series format. We wanted to do this for zeroing out the urgptr in the router phase. However, we were unable to have a second NFQUEUE running on the computer. We already had an NFQUEUE running that was looking for the responses from the command and control to take in malicious packets. We were using an NFQUEUE to send the response back in the TCP urgent header field back to the C&C after it would run its command.   There is a known bug in scapy that does not send packets over layer2 and thus we could not zero out the pointer before it was received by the web server.

To test the defense you need to run several programs:
first start the attack described above, then:

```
$ ssh cs4404@10.4.7.3
> sudo su
> ./run_barnyard2
```
Lastly, in cs4404@10.4.7.3 run:
```
> sudo su
# ./run_snort
```

After running this, you should see something that resembles the images in appendix B. For the attacking and receiving VMs, the output can be seen in figure 3 in appendix A.

Conclusion

The goal for this project was to be able to hide information in a covert channel, a place that data/information would normally not be stored in. We were able to successfully hide our messages in the Urgent Pointer header of the TCP packet and use the commands to gather information about the infected host. One beneficial skill we learned is the ability to write certain headers using Scapy. For our last project, we attempted to do something similar by hiding our DNS queries and resolves in different parts of the packet, however it was edited using DNS and not Scapy. Scapy allowed us to have a more clean edit of the packets than what we did last time. We went into extensive detail with Scapy and learning how to check to see if there was data in certain parts of packets and writing to different parts of the header.

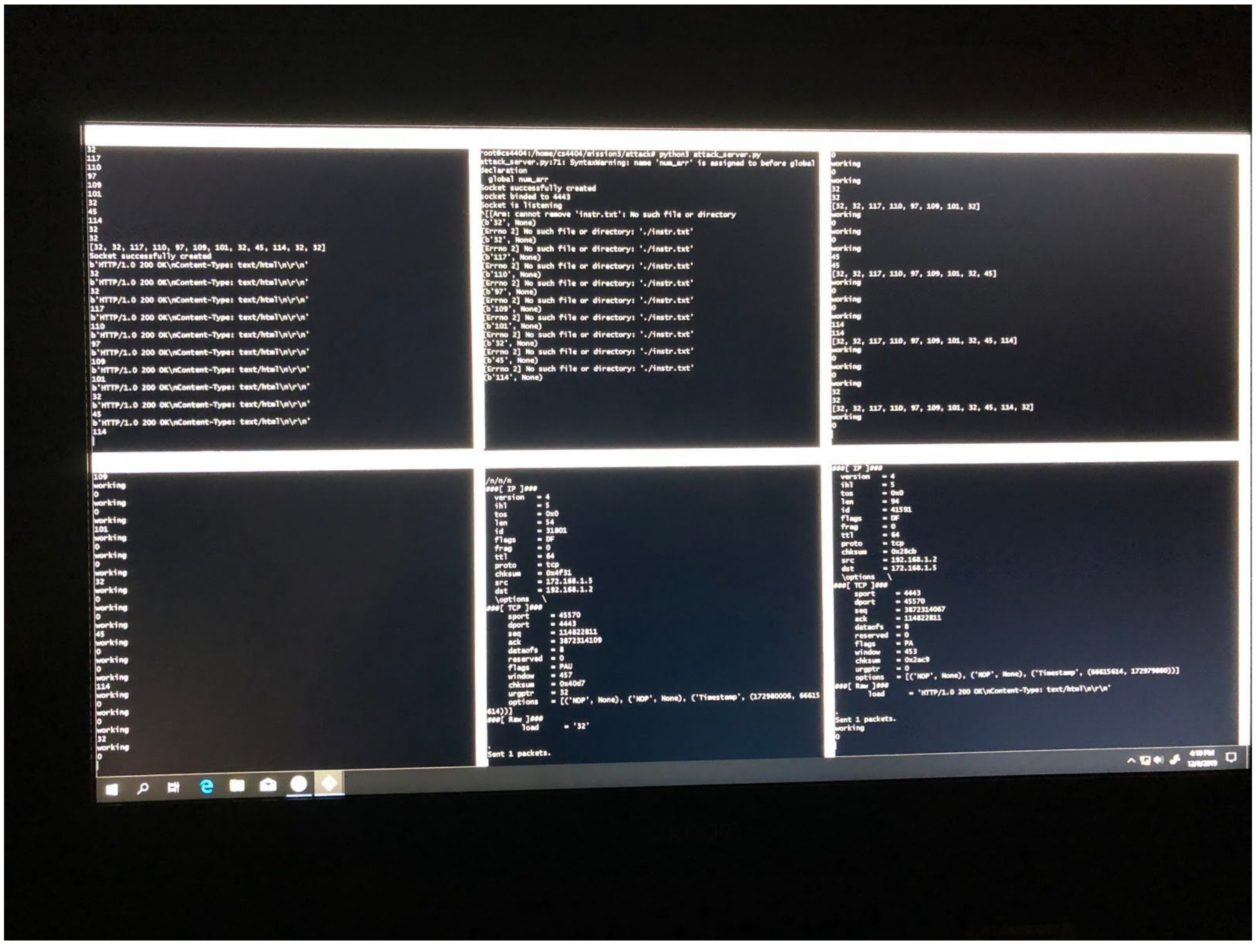
We were also to successfully learn Snort and rules to implement for it. We have learned from our attempt at attack that Snort is a lot stronger than we had initially thought. While attempting to flood Snort with packets, it was able to survive and stay up even though it was being bombarded. We were able to learn how to write very specific rules regarding the type of packets, the number of packets in a given time frame, as well as the contents of the packets, which we would be able to drop. We also learned how Snort interfaced with a database. We learned that when using Barnyard2, it would write to an ‘event’ table within a Snort database. We were then able to use Python to be able to make database calls, something that none of us have done before either.

We attempted to implement a defense that we eventually realized would not work the way that we thought it would. We initially attempted to use Barnyard2 and a MySQL database defense. We wrote a query to be able to set the database an updated time frame. Then this would allow us to check to see if there was X amount of packets being sent through the network after a given time by comparing the seconds. We wanted to have two NFQUEUEs on one machine, however, we were unable to do so. However, we ended up learning a lot in the process. The only thing that hindered our defense from working was the NFQUEUEs. We had all the infrastructure to be able to implement a NFQUEUE self-defense to do the work Snort after it got overloaded. We wanted to be able to have Snort detect a certain amount of packets with Python and send a message out to each of the machines that would then filter incoming packets and 0 out the urgent pointer. We would assume that you could 0 out other fields in the headers that you decided would be a possible place where information could be stored. However, having a NFQUEUE for the malicious server as well as an NFQUEUE for the defensive server was not working. Therefore, we decided to move to a defense that stopped the flooding of our system and attempted to look for any additional information in the Urgent Pointer field. Again, this is assuming that you can also look for any other parts of the TCP packet that you might hypothesize malicious information being stored.

We have learned quite a lot from this project including Snort, databases, in-depth knowledge of Scapy, and exploring the TCP packet headers themselves.

Appendix A

Attack phase output:



```
32
117
110
97
109
101
32
45
114
32
32
[32, 32, 117, 110, 97, 109, 101, 32, 45, 114, 32, 32]
Socket successfully created
b'HTTP/1.0 200 OK\nContent-Type: text/html\n\r\n'
32
b'HTTP/1.0 200 OK\nContent-Type: text/html\n\r\n'
32
b'HTTP/1.0 200 OK\nContent-Type: text/html\n\r\n'
117
b'HTTP/1.0 200 OK\nContent-Type: text/html\n\r\n'
110
b'HTTP/1.0 200 OK\nContent-Type: text/html\n\r\n'
97
b'HTTP/1.0 200 OK\nContent-Type: text/html\n\r\n'
109
b'HTTP/1.0 200 OK\nContent-Type: text/html\n\r\n'
101
b'HTTP/1.0 200 OK\nContent-Type: text/html\n\r\n'
32
b'HTTP/1.0 200 OK\nContent-Type: text/html\n\r\n'
45
b'HTTP/1.0 200 OK\nContent-Type: text/html\n\r\n'
114
```

```
root@cs4404:/home/cs4404/mission3/attack# python3 attack_server.py
attack_server.py:71: SyntaxWarning: name 'num_arr' is assigned to before global
declaration
  global num_arr
Socket successfully created
socket binded to 4443
Socket is listening
^[[Arm: cannot remove 'instr.txt': No such file or directory
(b'32', None)
[Errno 2] No such file or directory: './instr.txt'
(b'32', None)
[Errno 2] No such file or directory: './instr.txt'
(b'117', None)
[Errno 2] No such file or directory: './instr.txt'
(b'110', None)
[Errno 2] No such file or directory: './instr.txt'
(b'97', None)
[Errno 2] No such file or directory: './instr.txt'
(b'109', None)
[Errno 2] No such file or directory: './instr.txt'
(b'101', None)
[Errno 2] No such file or directory: './instr.txt'
(b'32', None)
[Errno 2] No such file or directory: './instr.txt'
(b'45', None)
[Errno 2] No such file or directory: './instr.txt'
(b'114', None)
```

```
0
working
0
working
32
32
[32, 32, 117, 110, 97, 109, 101, 32]
working
0
working
0
working
45
45
[32, 32, 117, 110, 97, 109, 101, 32, 45]
working
0
working
0
working
114
114
[32, 32, 117, 110, 97, 109, 101, 32, 45, 114]
working
0
working
0
working
32
32
[32, 32, 117, 110, 97, 109, 101, 32, 45, 114, 32]
working
0
```

```
109
working
0
working
0
working
101
working
0
working
0
working
32
working
0
working
0
working
45
working
0
working
0
working
114
working
0
working
0
working
32
working
0
```

```
/n/n/n
###[ IP ]###
  version   = 4
  ihl       = 5
  tos       = 0x0
  len       = 54
  id        = 31801
  flags     = DF
  frag      = 0
  ttl       = 64
  proto     = tcp
  chksum    = 0x4f31
  src       = 172.168.1.5
  dst       = 192.168.1.2
  \options   \
###[ TCP ]###
     sport     = 45570
     dport     = 4443
     seq       = 114822811
     ack       = 3872314109
     dataofs   = 8
     reserved  = 0
     flags     = PAU
     window    = 457
     chksum    = 0x40d7
     urgptr    = 32
     options   = [('NOP', None), ('NOP', None), ('Timestamp', (172980006, 66615
614))]
###[ Raw ]###
        load      = '32'

.
Sent 1 packets.
```

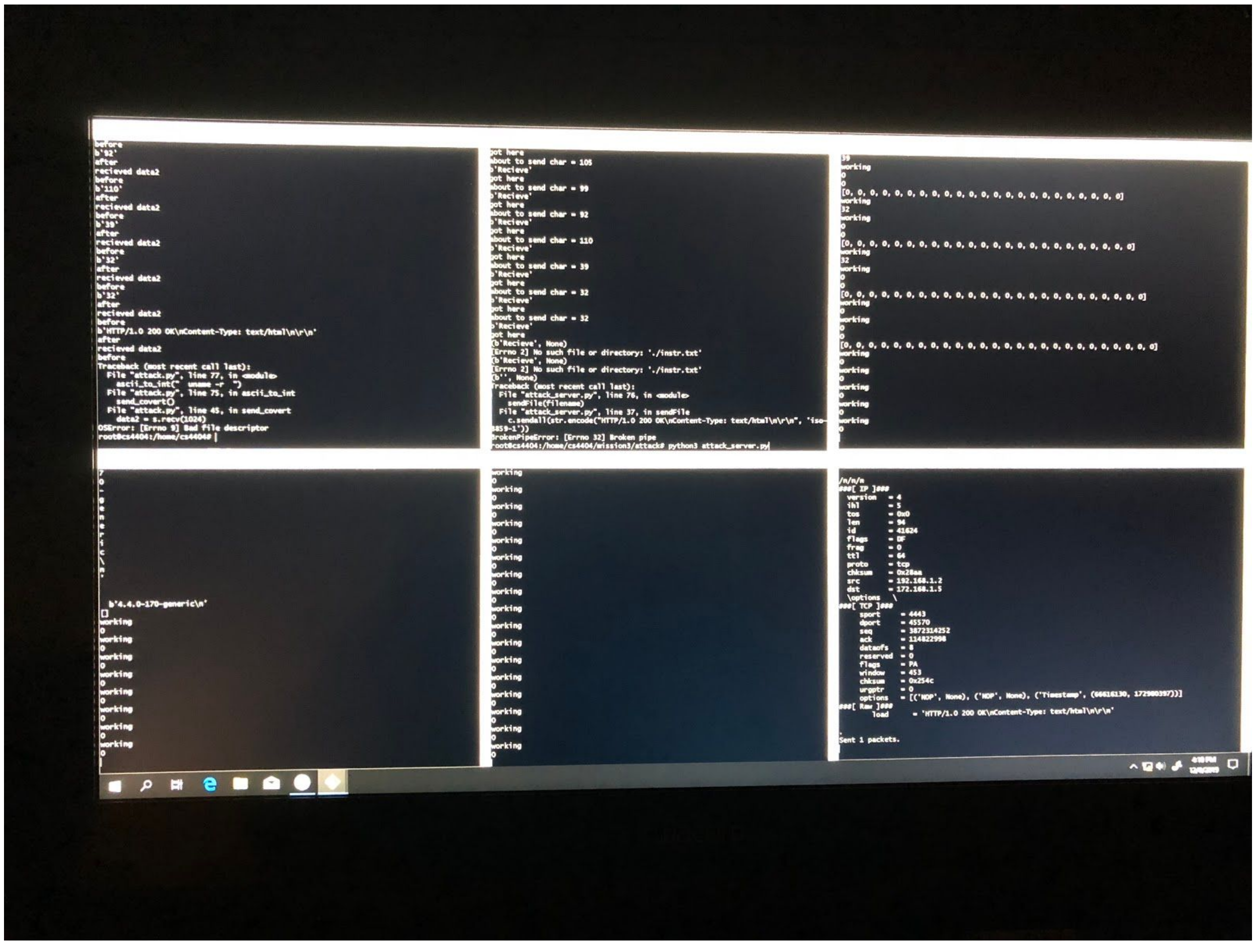
```
###[ IP ]###
  version   = 4
  ihl       = 5
  tos       = 0x0
  len       = 94
  id        = 41591
  flags     = DF
  frag      = 0
  ttl       = 64
  proto     = tcp
  chksum    = 0x28cb
  src       = 192.168.1.2
  dst       = 172.168.1.5
  \options   \
###[ TCP ]###
     sport     = 4443
     dport     = 45570
     seq       = 3872314067
     ack       = 114822811
     dataofs   = 8
     reserved  = 0
     flags     = PA
     window    = 453
     chksum    = 0x2ac9
     urgptr    = 0
     options   = [('NOP', None), ('NOP', None), ('Timestamp', (66615614, 172979880))]
###[ Raw ]###
        load      = 'HTTP/1.0 200 OK\nContent-Type: text/html\n\r\n'

.
Sent 1 packets.
working
0
```

Figure 1: Start of Attack

(top left: attack.py, top middle: attack_server.py, top right: server_scapy_test.py, bottom left: instruction_response.py, bottom middle: queue_edit.py, bottom right: queue_exit_response.py)

In the image above it shows the attack run from the top left. You can see the values being sent out followed by the values sent back to it. In the top middle image you can see the values being received. In the top right you can see the values from C&C being compiled in an array. After the array is completed it will be put in a file that can be read in the top middle. The other three currely show debug output for sent and received output.



```
before
b'92'
after
recieved data2
before
b'110'
after
recieved data2
before
b'39'
after
recieved data2
before
b'32'
after
recieved data2
before
b'32'
after
recieved data2
before
b'HTTP/1.0 200 OK\nContent-Type: text/html\n\r\n'
after
recieved data2
before
Traceback (most recent call last):
  File "attack.py", line 77, in <module>
    ascii_to_int("  uname -r  ")
  File "attack.py", line 75, in ascii_to_int
    send_covert()
  File "attack.py", line 45, in send_covert
    data2 = s.recv(1024)
OSError: [Errno 9] Bad file descriptor
root@cs4404:/home/cs4404# |
```

```
got here
about to send char = 105
b'Recieve'
got here
about to send char = 99
b'Recieve'
got here
about to send char = 92
b'Recieve'
got here
about to send char = 110
b'Recieve'
got here
about to send char = 39
b'Recieve'
got here
about to send char = 32
b'Recieve'
got here
about to send char = 32
b'Recieve'
got here
(b'Recieve', None)
[Errno 2] No such file or directory: './instr.txt'
(b'Recieve', None)
[Errno 2] No such file or directory: './instr.txt'
(b'', None)
Traceback (most recent call last):
  File "attack_server.py", line 76, in <module>
    sendFile(filename)
  File "attack_server.py", line 37, in sendFile
    c.sendall(str.encode("HTTP/1.0 200 OK\nContent-Type: text/html\n\r\n", 'iso-
8859-1'))
BrokenPipeError: [Errno 32] Broken pipe
root@cs4404:/home/cs4404/mission3/attack# python3 attack_server.py
```

```
39
working
0
0
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
working
32
working
0
0
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
working
32
working
0
0
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
working
0
working
0
0
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
working
0
working
0
working
0
working
0
working
0
```

```
7
0
-
g
e
n
e
r
i
c
\
n
'



 b'4.4.0-170-generic\n'
[]
working
0
working
0
working
0
working
0
working
0
working
0
working
0
working
0
```

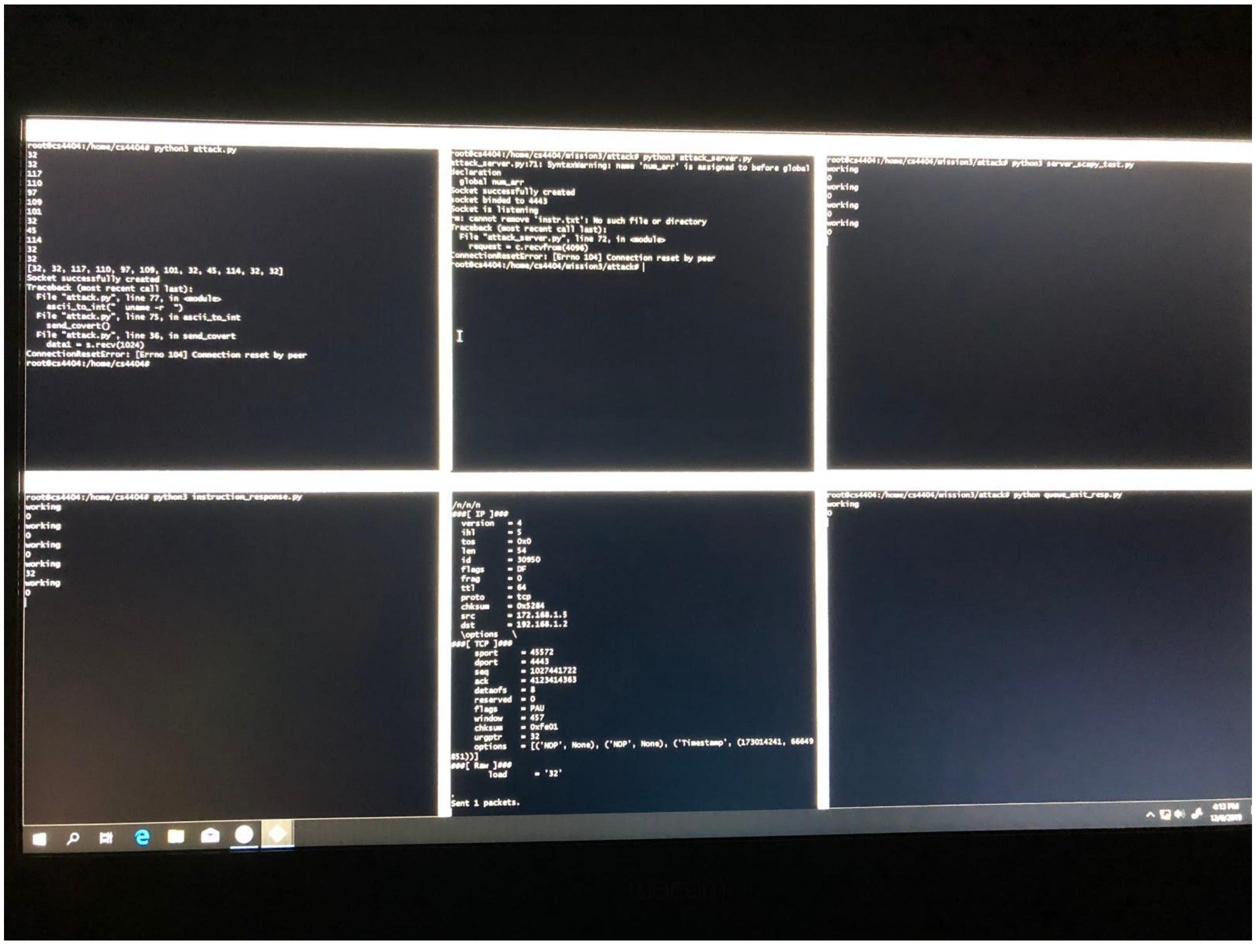
```
working
0
working
0
working
0
working
0
working
0
working
0
working
0
working
0
working
0
working
0
working
0
working
0
working
0
working
0
working
0
working
0
working
0
```

```
/n/n/n
###[ IP ]###
  version   = 4
  ihl       = 5
  tos       = 0x0
  len       = 94
  id        = 41624
  flags     = DF
  frag      = 0
  ttl       = 64
  proto     = tcp
  chksum    = 0x28aa
  src       = 192.168.1.2
  dst       = 172.168.1.5
  \options   \
###[ TCP ]###
     sport     = 4443
     dport     = 45570
     seq       = 3872314252
     ack       = 114822998
     dataofs   = 8
     reserved  = 0
     flags     = PA
     window    = 453
     chksum    = 0x254c
     urgptr    = 0
     options   = [('NOP', None), ('NOP', None), ('Timestamp', (66616130, 172980397))]
###[ Raw ]###
        load      = 'HTTP/1.0 200 OK\nContent-Type: text/html\n\r\n'

.
Sent 1 packets.
```

Figure 2: End of Attack
(top left: attack.py, top middle: attack_server.py, top right: server_scapy_test.py, bottom left: instruction_response.py, bottom middle: queue_edit.py, bottom right: queue_exit_response.py)

In the image above you can see the top right cancels the connection after it has received the output of the instruction sent from the corrupted server. The top middle vm has the same type of output. It shows the values it is sending then the result of the cancelled connection. The bottom right vm has the program that receives the output from the attack. It prints out the values it receives then the combined string after it is translated to hex.



```
root@cs4404:/home/cs4404# python3 attack.py
32
32
117
110
97
109
101
32
45
114
32
32
[32, 32, 117, 110, 97, 109, 101, 32, 45, 114, 32, 32]
Socket successfully created
Traceback (most recent call last):
  File "attack.py", line 77, in <module>
    ascii_to_int("  uname -r  ")
  File "attack.py", line 75, in ascii_to_int
    send_covert()
  File "attack.py", line 36, in send_covert
    data1 = s.recv(1024)
ConnectionResetError: [Errno 104] Connection reset by peer
root@cs4404:/home/cs4404#
```

```
root@cs4404:/home/cs4404/mission3/attack# python3 attack_server.py
attack_server.py:71: SyntaxWarning: name 'num_arr' is assigned to before global
declaration
  global num_arr
Socket successfully created
socket binded to 4443
Socket is listening
rm: cannot remove 'instr.txt': No such file or directory
Traceback (most recent call last):
  File "attack_server.py", line 72, in <module>
    request = c.recvfrom(4096)
ConnectionResetError: [Errno 104] Connection reset by peer
root@cs4404:/home/cs4404/mission3/attack# |
```

```
root@cs4404:/home/cs4404/mission3/attack# python3 server_scapy_test.py
working
0
working
0
working
0
working
0
```

```
root@cs4404:/home/cs4404# python3 instruction_response.py
working
0
working
0
working
0
working
32
working
0
```

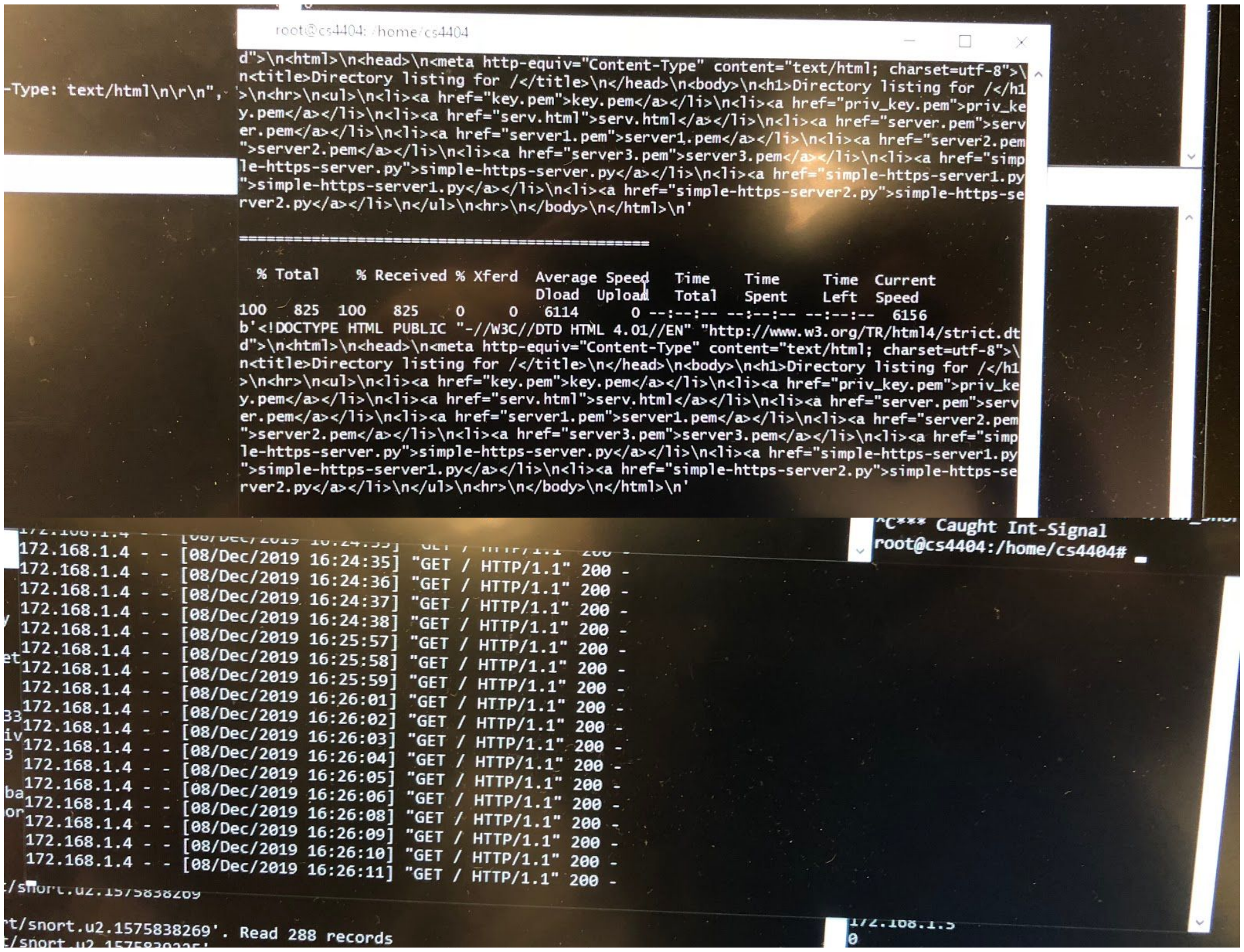
```
/n/n/n
###[ IP ]###
  version   = 4
  ihl       = 5
  tos       = 0x0
  len       = 54
  id        = 30950
  flags     = DF
  frag      = 0
  ttl       = 64
  proto     = tcp
  chksum    = 0x5284
  src       = 172.168.1.5
  dst       = 192.168.1.2
  \options   \
###[ TCP ]###
     sport     = 45572
     dport     = 4443
     seq       = 1027441722
     ack       = 4123414363
     dataofs   = 8
     reserved  = 0
     flags     = PAU
     window    = 457
     chksum    = 0xfe01
     urgptr    = 32
     options   = [('NOP', None), ('NOP', None), ('Timestamp', (173014241, 66649
851))]
###[ Raw ]###
        load      = '32'

.
Sent 1 packets.
```

```
root@cs4404:/home/cs4404/mission3/attack# python queue_exit_resp.py
working
0
```

Figure 3: End of Defense
(top left: attack.py, top middle: attack_server.py, top right: server_scapy_test.py, bottom left: instruction_response.py, bottom middle: queue_edit.py, bottom right: queue_exit_response.py)

In this image it shows that the defence cancels the connection between the C&C and corrupted server at port 4443. If you run the infrastructure at the same time it will not be blocked.



```
root@cs4404: /home/cs4404
-Type: text/html\n\r\n",
d">\n<html>\n<head>\n<meta http-equiv="Content-Type" content="text/html; charset=utf-8">\
n<title>Directory listing for /</title>\n</head>\n<body>\n<h1>Directory listing for /</h1
>\n<hr>\n<ul>\n<li><a href="key.pem">key.pem</a></li>\n<li><a href="priv_key.pem">priv_ke
y.pem</a></li>\n<li><a href="serv.html">serv.html</a></li>\n<li><a href="server.pem">serv
er.pem</a></li>\n<li><a href="server1.pem">server1.pem</a></li>\n<li><a href="server2.pem
">server2.pem</a></li>\n<li><a href="server3.pem">server3.pem</a></li>\n<li><a href="simp
le-https-server.py">simple-https-server.py</a></li>\n<li><a href="simple-https-server1.py
">simple-https-server1.py</a></li>\n<li><a href="simple-https-server2.py">simple-https-se
rver2.py</a></li>\n</ul>\n<hr>\n</body>\n</html>\n'

==================================================

  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   825  100   825    0     0   6114      0 --:--:-- --:--:-- --:--:--  6156
b'<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dt
d">\n<html>\n<head>\n<meta http-equiv="Content-Type" content="text/html; charset=utf-8">\
n<title>Directory listing for /</title>\n</head>\n<body>\n<h1>Directory listing for /</h1
>\n<hr>\n<ul>\n<li><a href="key.pem">key.pem</a></li>\n<li><a href="priv_key.pem">priv_ke
y.pem</a></li>\n<li><a href="serv.html">serv.html</a></li>\n<li><a href="server.pem">serv
er.pem</a></li>\n<li><a href="server1.pem">server1.pem</a></li>\n<li><a href="server2.pem
">server2.pem</a></li>\n<li><a href="server3.pem">server3.pem</a></li>\n<li><a href="simp
le-https-server.py">simple-https-server.py</a></li>\n<li><a href="simple-https-server1.py
">simple-https-server1.py</a></li>\n<li><a href="simple-https-server2.py">simple-https-se
rver2.py</a></li>\n</ul>\n<hr>\n</body>\n</html>\n'
```

```
172.168.1.4 - - [08/Dec/2019 16:24:35] "GET / HTTP/1.1" 200 -
172.168.1.4 - - [08/Dec/2019 16:24:36] "GET / HTTP/1.1" 200 -
172.168.1.4 - - [08/Dec/2019 16:24:37] "GET / HTTP/1.1" 200 -
172.168.1.4 - - [08/Dec/2019 16:24:38] "GET / HTTP/1.1" 200 -
172.168.1.4 - - [08/Dec/2019 16:25:57] "GET / HTTP/1.1" 200 -
172.168.1.4 - - [08/Dec/2019 16:25:58] "GET / HTTP/1.1" 200 -
172.168.1.4 - - [08/Dec/2019 16:25:59] "GET / HTTP/1.1" 200 -
172.168.1.4 - - [08/Dec/2019 16:26:01] "GET / HTTP/1.1" 200 -
172.168.1.4 - - [08/Dec/2019 16:26:02] "GET / HTTP/1.1" 200 -
172.168.1.4 - - [08/Dec/2019 16:26:03] "GET / HTTP/1.1" 200 -
172.168.1.4 - - [08/Dec/2019 16:26:04] "GET / HTTP/1.1" 200 -
172.168.1.4 - - [08/Dec/2019 16:26:05] "GET / HTTP/1.1" 200 -
172.168.1.4 - - [08/Dec/2019 16:26:06] "GET / HTTP/1.1" 200 -
172.168.1.4 - - [08/Dec/2019 16:26:08] "GET / HTTP/1.1" 200 -
172.168.1.4 - - [08/Dec/2019 16:26:09] "GET / HTTP/1.1" 200 -
172.168.1.4 - - [08/Dec/2019 16:26:10] "GET / HTTP/1.1" 200 -
172.168.1.4 - - [08/Dec/2019 16:26:11] "GET / HTTP/1.1" 200 -

t/snort.u2.1575838269'. Read 288 records
```

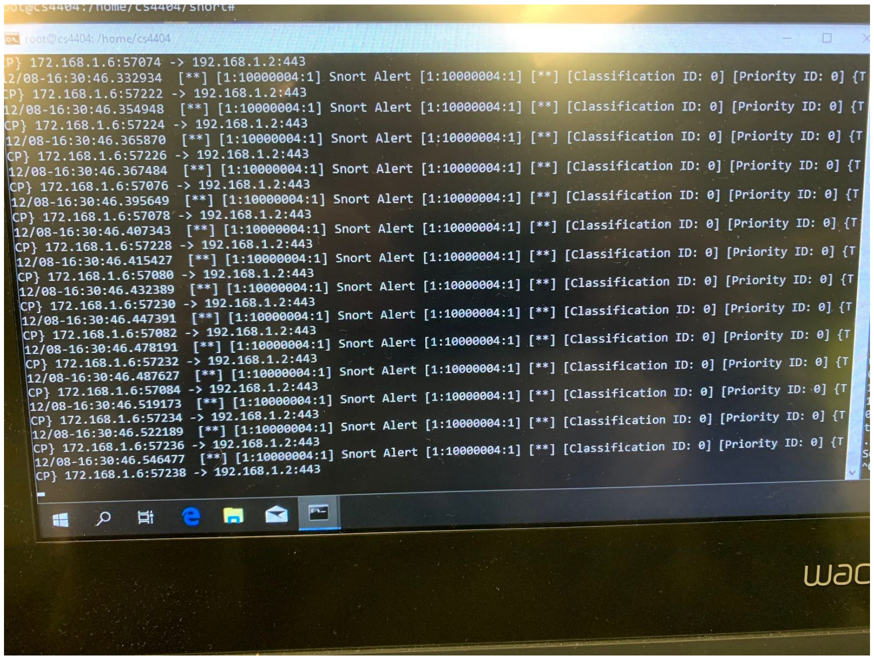
```
^C*** Caught Int-Signal
root@cs4404:/home/cs4404# _
```

Figure 4: Real Traffic

(Top: real_traffic.py, Bottom: simple-https-server2.py)

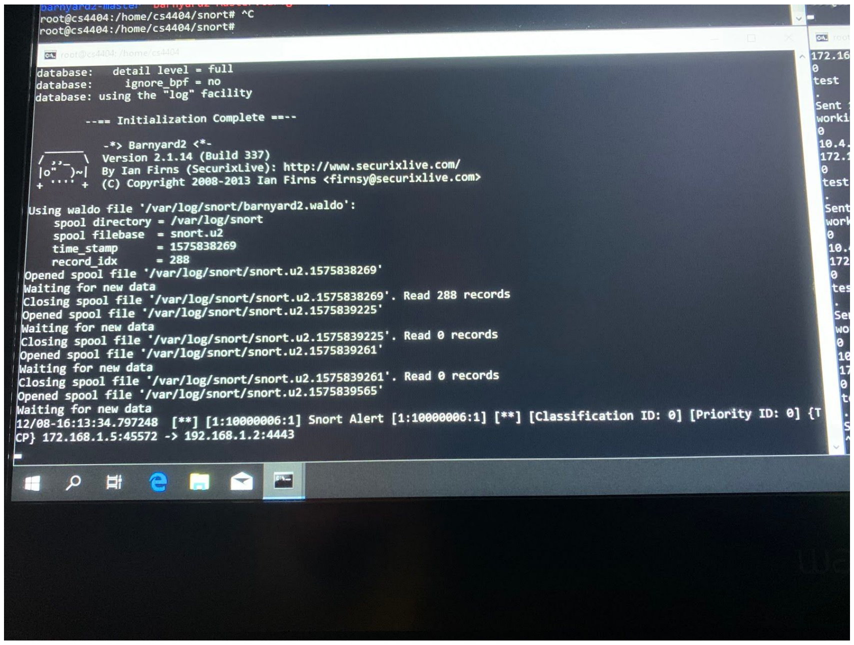
In this figure the bottom image shows that the https requests were successful as it has a 200 output. The image above it shows that html is returned successfully.

Appendix B:



```
root@cs4404: /home/cs4404
P} 172.168.1.6:57074 -> 192.168.1.2:443
2/08-16:30:46.332934  [**] [1:10000004:1] Snort Alert [1:10000004:1] [**] [Classification ID: 0] [Priority ID: 0] {T
P} 172.168.1.6:57222 -> 192.168.1.2:443
12/08-16:30:46.354948  [**] [1:10000004:1] Snort Alert [1:10000004:1] [**] [Classification ID: 0] [Priority ID: 0] {T
CP} 172.168.1.6:57224 -> 192.168.1.2:443
12/08-16:30:46.365870  [**] [1:10000004:1] Snort Alert [1:10000004:1] [**] [Classification ID: 0] [Priority ID: 0] {T
CP} 172.168.1.6:57226 -> 192.168.1.2:443
12/08-16:30:46.367484  [**] [1:10000004:1] Snort Alert [1:10000004:1] [**] [Classification ID: 0] [Priority ID: 0] {T
CP} 172.168.1.6:57076 -> 192.168.1.2:443
12/08-16:30:46.395649  [**] [1:10000004:1] Snort Alert [1:10000004:1] [**] [Classification ID: 0] [Priority ID: 0] {T
CP} 172.168.1.6:57078 -> 192.168.1.2:443
12/08-16:30:46.407343  [**] [1:10000004:1] Snort Alert [1:10000004:1] [**] [Classification ID: 0] [Priority ID: 0] {T
CP} 172.168.1.6:57228 -> 192.168.1.2:443
12/08-16:30:46.415427  [**] [1:10000004:1] Snort Alert [1:10000004:1] [**] [Classification ID: 0] [Priority ID: 0] {T
CP} 172.168.1.6:57080 -> 192.168.1.2:443
12/08-16:30:46.432389  [**] [1:10000004:1] Snort Alert [1:10000004:1] [**] [Classification ID: 0] [Priority ID: 0] {T
CP} 172.168.1.6:57230 -> 192.168.1.2:443
12/08-16:30:46.447391  [**] [1:10000004:1] Snort Alert [1:10000004:1] [**] [Classification ID: 0] [Priority ID: 0] {T
CP} 172.168.1.6:57082 -> 192.168.1.2:443
12/08-16:30:46.478191  [**] [1:10000004:1] Snort Alert [1:10000004:1] [**] [Classification ID: 0] [Priority ID: 0] {T
CP} 172.168.1.6:57232 -> 192.168.1.2:443
12/08-16:30:46.487627  [**] [1:10000004:1] Snort Alert [1:10000004:1] [**] [Classification ID: 0] [Priority ID: 0] {T
CP} 172.168.1.6:57084 -> 192.168.1.2:443
12/08-16:30:46.519173  [**] [1:10000004:1] Snort Alert [1:10000004:1] [**] [Classification ID: 0] [Priority ID: 0] {T
CP} 172.168.1.6:57234 -> 192.168.1.2:443
12/08-16:30:46.522189  [**] [1:10000004:1] Snort Alert [1:10000004:1] [**] [Classification ID: 0] [Priority ID: 0] {T
CP} 172.168.1.6:57236 -> 192.168.1.2:443
12/08-16:30:46.546477  [**] [1:10000004:1] Snort Alert [1:10000004:1] [**] [Classification ID: 0] [Priority ID: 0] {T
CP} 172.168.1.6:57238 -> 192.168.1.2:443
```

- This image indicates Snort alerted and blocking the attempted flood against the web server. Once the server received over 70 packets in 10 seconds from a single IP that IP was then blocked until it stopped sending packets.



```
root@cs4404:/home/cs4404/snort# ^C
root@cs4404:/home/cs4404/snort#

root@cs4404: /home/cs4404
database:   detail level = full
database:     ignore_bpf = no
database: using the "log" facility

        --== Initialization Complete ==--

            -*> Barnyard2 <*-
 / ,,_  \   Version 2.1.14 (Build 337)
 |o"  )~|   By Ian Firns (SecurixLive): http://www.securixlive.com/
 + '''' +   (C) Copyright 2008-2013 Ian Firns <firnsy@securixlive.com>

Using waldo file '/var/log/snort/barnyard2.waldo':
    spool directory = /var/log/snort
    spool filebase  = snort.u2
    time_stamp      = 1575838269
    record_idx      = 288
Opened spool file '/var/log/snort/snort.u2.1575838269'
Waiting for new data
Closing spool file '/var/log/snort/snort.u2.1575838269'. Read 288 records
Opened spool file '/var/log/snort/snort.u2.1575839225'
Waiting for new data
Closing spool file '/var/log/snort/snort.u2.1575839225'. Read 0 records
Opened spool file '/var/log/snort/snort.u2.1575839261'
Waiting for new data
Closing spool file '/var/log/snort/snort.u2.1575839261'. Read 0 records
Opened spool file '/var/log/snort/snort.u2.1575839565'
Waiting for new data
12/08-16:13:34.797248  [**] [1:10000006:1] Snort Alert [1:10000006:1] [**] [Classification ID: 0] [Priority ID: 0] {T
CP} 172.168.1.5:45572 -> 192.168.1.2:4443
```

- This image indicates Snort dropping the packet that has the PAU flag.

Citations: APA format

1. What is deep packet analysis (and the 7 best tools to use). (2019, August 29). Retrieved from https://www.comparitech.com/net-admin/deep-packet-inspection/.
2. Conklin, K. (2018, August 20). What Is Network Flow Monitoring? Retrieved from https://www.whatsupgold.com/blog/network-monitoring/what-is-network-flow-monitoring/.
3. What is NetFlow? How NetFlow Works & How to Use. (n.d.). Retrieved from https://www.solarwinds.com/netflow-traffic-analyzer/use-cases/what-is-netflow.
4. NetFlow Advantages in a Nutshell. (2016, December 28). Retrieved December 8, 2019, from https://www.plixer.com/blog/netflow-advantages-in-a-nutshell/
5. McIntyre, J. (2015, July 7). Using Snort for intrusion detection. Retrieved December 8, 2019, from https://www.techrepublic.com/article/using-snort-for-intrusion-detection/
6. NetFlow vs. sFlow vs. IPFIX vs. NetStream. Network Traffic Analysis and Network Traffic Monitoring. (2018, November 19). Retrieved December 8, 2019, from https://www.noction.com/blog/netflow-sflow-ipfix-netstream
7. Burrill, J. (2015, September 27). What Is Deep Packet Inspection And Its Advantages and Disadvantages? Retrieved December 8, 2019, from http://bloggerspath.com/what-is-deep-packet-inspection-and-its-advantages-and-disadvantages/
8. Server Resolution Error 1001. (2019, August 1). Retrieved December 8, 2019, from https://www.sdxcentral.com/networking/sdn/definitions/what-is-openflow/
9. CAS – Central Authentication Service. (et al.). Retrieved December 8, 2019, from https://cas.wpi.edu/cas/login?service=https%3a%2f%2fia.wpi.edu%2fwpi_cas_auth%2f
10. MacVittie, L. (2019, January 10). The Problem With SDN. Retrieved December 8, 2019, from https://www.networkcomputing.com/networking/problem-sdn
11. Deep Packet Inspection Tool - Analysis Software | SolarWinds. (n.d.). Retrieved December 8, 2019, from https://www.solarwinds.com/network-performance-monitor/use-cases/deep-packet-inspection