Luis Medina

CS4351

Due 11/29/20

Assignment 5 – Snort Network IDS

Problem

For this assignment we were to create rules for Snort to defend against the various TCP attacks we performed in assignment 4. We were to use the same environment of a local network using three virtual machines. One of these VMs was a host, another was a client and the last was the attacker.

Setup

1. First, I cloned my original VM and created three others.
2. Next, I set their networking type to "Internal Network" and enabled full promiscuous mode.
3. I then set static IPs for the VMs:
    a. Client: 10.0.2.15
    b. Server (Victim): 10.0.2.16
    c. Attacker: 10.0.2.17
4. Installed Snort on Server machine with: **sudo apt install snort**
    a. Set up to use **enp0s3** and **10.0.2.16/24**
    b. Configured HOME_NET variable to **[10.0.2.15/24, 10.0.2.16/24]**
5. I disabled tcp syn cookies on the host machine by running the following command: **sudo sysctl -w net.ipvv4.tcp_syncookies=0**

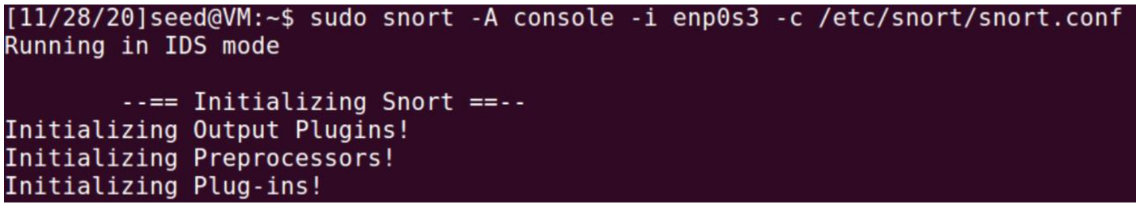
Task 1 – Detecting and Alerting SYN Flooding Attack

1. First, I had to determine what to look for in packets to detect a SYN flooding attack. What makes a SYN flooding attack dangerous is its numerous SYN requests. This tells me two things: I need to look for "S" flags in packets and I have to detect several of them. Snort provides the option **detection_filter** that allows a rule to count a number of packets in a specified time in seconds. It, however, needs to track a packet by an IP. Since the source IPs of the packets are spoofed, I set the rule to track them by the destination. Resulting in the rule:

    alert tcp any any -> any 23 (msg:"Possible SYN Flood"; detection_filter:track by_dst, count 4, seconds 15; flags:S,CE; sid:1000001;)

    For this assignment, I set the count to a low number in a large time window to test the rule. This was perfect as I knew the only connection to the server would be that of the attacker for this portion. In a real-life setting, these numbers would need to be adjusted depending on how many connections the server would be expected to see.

2. Testing rule in the environment
    a. I set the rule in the snort.conf file
    b. Started snort on the server with: **sudo snort -A console -i enp0s3-c /etc/snort/snort.conf** (*Image 1*)

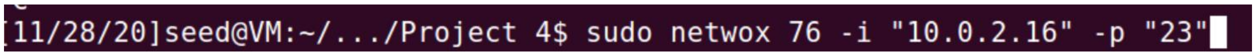


```
[11/28/20]seed@VM:~$ sudo snort -A console -i enp0s3 -c /etc/snort/snort.conf
Running in IDS mode

        --== Initializing Snort ==--
Initializing Output Plugins!
Initializing Preprocessors!
Initializing Plug-ins!
```

Image 1

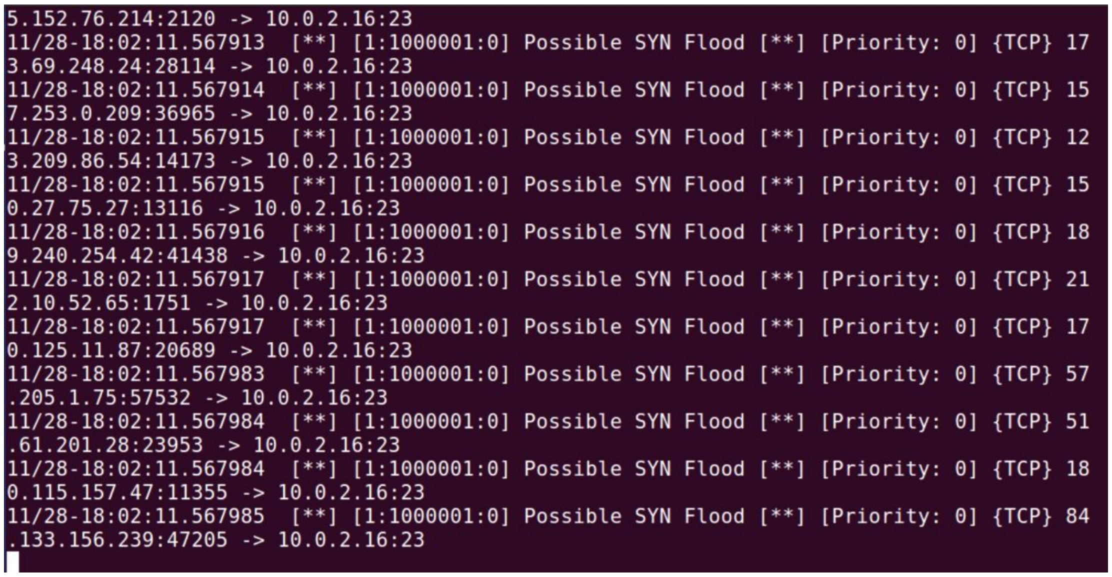
    c. I then ran the attack on the attacker VM (*image 2*)



```
[11/28/20]seed@VM:~/.../Project 4$ sudo netwox 76 -i "10.0.2.16" -p "23"
```

Image 2

    d. This resulted in many alerts (*image 3*)



```
5.152.76.214:2120 -> 10.0.2.16:23
11/28-18:02:11.567913  [**] [1:1000001:0] Possible SYN Flood [**] [Priority: 0] {TCP} 17
3.69.248.24:28114 -> 10.0.2.16:23
11/28-18:02:11.567914  [**] [1:1000001:0] Possible SYN Flood [**] [Priority: 0] {TCP} 15
7.253.0.209:36965 -> 10.0.2.16:23
11/28-18:02:11.567915  [**] [1:1000001:0] Possible SYN Flood [**] [Priority: 0] {TCP} 12
3.209.86.54:14173 -> 10.0.2.16:23
11/28-18:02:11.567915  [**] [1:1000001:0] Possible SYN Flood [**] [Priority: 0] {TCP} 15
0.27.75.27:13116 -> 10.0.2.16:23
11/28-18:02:11.567916  [**] [1:1000001:0] Possible SYN Flood [**] [Priority: 0] {TCP} 18
9.240.254.42:41438 -> 10.0.2.16:23
11/28-18:02:11.567917  [**] [1:1000001:0] Possible SYN Flood [**] [Priority: 0] {TCP} 21
2.10.52.65:1751 -> 10.0.2.16:23
11/28-18:02:11.567917  [**] [1:1000001:0] Possible SYN Flood [**] [Priority: 0] {TCP} 17
0.125.11.87:20689 -> 10.0.2.16:23
11/28-18:02:11.567983  [**] [1:1000001:0] Possible SYN Flood [**] [Priority: 0] {TCP} 57
.205.1.75:57532 -> 10.0.2.16:23
11/28-18:02:11.567984  [**] [1:1000001:0] Possible SYN Flood [**] [Priority: 0] {TCP} 51
.61.201.28:23953 -> 10.0.2.16:23
11/28-18:02:11.567984  [**] [1:1000001:0] Possible SYN Flood [**] [Priority: 0] {TCP} 18
0.115.157.47:11355 -> 10.0.2.16:23
11/28-18:02:11.567985  [**] [1:1000001:0] Possible SYN Flood [**] [Priority: 0] {TCP} 84
.133.156.239:47205 -> 10.0.2.16:23
```

image 3

Note: For this part, it would be possible to use event filters to minimize the amount of alerts. I did not implement it as the goal was to show the rule detecting the attack.

Task 2 – Detecting and Alerting about TCP RST Attacks on telnet Connections

1. For this part, I went through many options to try and minimize the amount of false alerts. I realized that a system receiving a packet, must take the information in the packet at face value. This means that it cannot tell if it contains a spoofed IP by simply looking at it. The only way for snort to detect the spoofed IP would be by checking the history of both legit parties and comparing the spoofed packet to it. Another way could be to check the MAC addresses of both parties to compare against the spoofed packet. This was something I was unable to get working. The closest I got was by designing a delay with the following rules:

   activate tcp any any -> any 23 (flags:R,CE; sid:1000002; activates: 1)

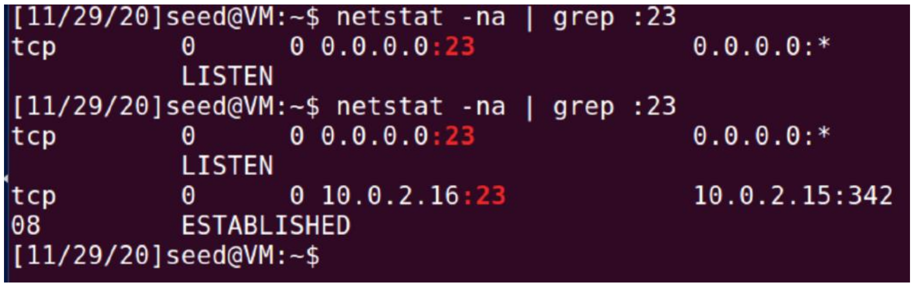
   dynamic tcp any any -> any 23 (msg:"Possible RST Attack"; detction_filter:track by_dst, count 1, seconds 10;)

   sdrop tcp any any -> any 23 (flags:R,CE; sid:1000003;)

   The idea was to have a rule detect any packet with the RST flag which would activate a second rule that would wait a predetermined amount of time. If it detected any traffic from the destination IP, it would mean the connection was not lost and would then send the alert of the RST attack. If no traffic was seen during the time window, it would most likely mean that the connection was indeed lost so the rule would terminate the session. Immediately after the activate rule fired, the third rule would see also see this packet and drop it, preventing the connection termination and allowing the timer to work. I was close but could not implement it in the end. One reason was due to not being able to chain multiple rules. Using the activation/dynamic method, it is only possible to chain two rules. Even then, the biggest issue was that my logic tries to use snort in a way that it was not intended. That is to wait for no traffic. Snort is meant to detect traffic and count or track multiple packages if needed, not to wait for no traffic before activating a rule.

2. In the end I used the following rule: **alert tcp any any -> any 23 (msg:"Possible RST Attack"; flags:R,CE; sid:1000005;)**

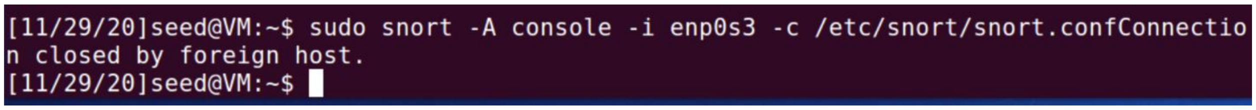
   a. I first connected the client to the server using **telnet 10.0.2.16** and verified the connection (*Image 4*)



```
[11/29/20]seed@VM:~$ netstat -na | grep :23
tcp        0      0 0.0.0.0:23              0.0.0.0:*
           LISTEN
[11/29/20]seed@VM:~$ netstat -na | grep :23
tcp        0      0 0.0.0.0:23              0.0.0.0:*
           LISTEN
tcp        0      0 10.0.2.16:23            10.0.2.15:342
08         ESTABLISHED
[11/29/20]seed@VM:~$
```

Image 4

b. Started snort on the server with: **sudo snort -A console -i enp0s3-c /etc/snort/snort.conf**
c. I then ran the attack on the attacker machine using the command **sudo netwox 78 -i 10.0.2.16**
d. In the end, the connection was dropped with no alert on the server (*Image 5*)



```
[11/29/20]seed@VM:~$ sudo snort -A console -i enp0s3 -c /etc/snort/snort.confConnectio
n closed by foreign host.
[11/29/20]seed@VM:~$ █
```

Image 5

This implementation would bring up false alarms for legit reset packets, but I figured it would be better to be notified for false positives than to not detect any attacks at all. I also attempted to use drop instead of alert to try and protect the connection but it made no difference.

Task 3 – Detecting and Alerting TCP Session Hijacking Attacks

1. For this rule, I decided to detect the attack by inspecting packet payloads. Initially I wanted to create a text file with a list of bash commands to check against using the "content-list" option in snort. I found out later that this option is now obsolete and no longer available. Instead, after using other rule files included with snort as an example, I decided to create multiple rules to detect these commands. It would also be possible to have several content check option in one rule but I decided that would hurt readability. The rules I came up with do have a chance of detecting false alarms as they might detect, for example, the command "cat" in a legit packet that is sending text. To mitigate this, I decided to look for a list of common bash commands and pick the ones that would be potentially dangerous when used in this attack, and then created rules for each of them. In the following example, I will be showing one of these rules:

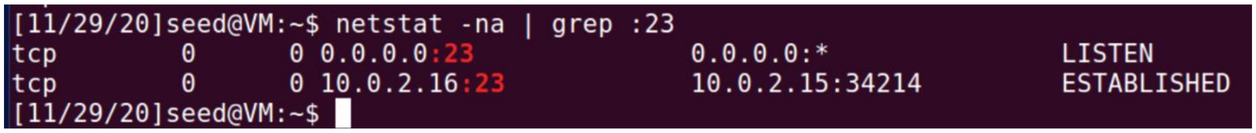
   alert tcp any any -> HOME_NET 23 (content: "|63 61 74|"; msg: "Session Hijack Attempt"; react: block; sid:1000006)

   The commands I found to be potentially dangerous:

   - cat
   - cp
   - mv
   - rm
   - touch
   - kill
   - killall
   - env
   - export

Note: The rule checks for "cat" in hex (636174) .

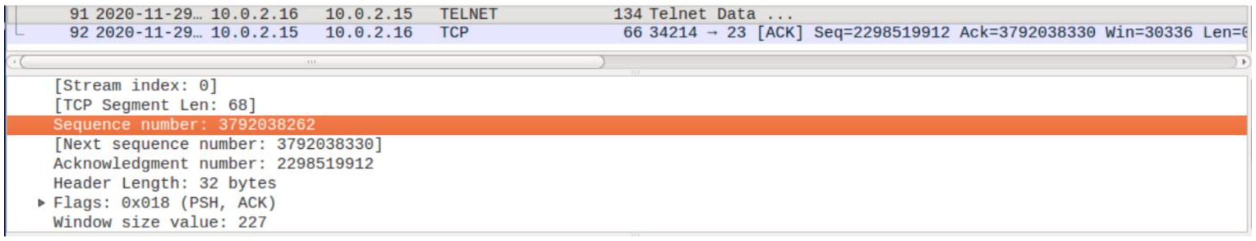
2. Testing the rule in the environment
    a. First, I opened wireshark on the attacker VM and set it to listen on **enp0s3**
    b. I then connected the client to the server using **telnet 10.0.2.16** and verified the connection (*image 6*)



```
[11/29/20]seed@VM:~$ netstat -na | grep :23
tcp        0      0 0.0.0.0:23              0.0.0.0:*               LISTEN
tcp        0      0 10.0.2.16:23            10.0.2.15:34214         ESTABLISHED
[11/29/20]seed@VM:~$
```

Image 6

    c. I then looked for the latest telnet packet seen by wireshark on the attacker VM (*Image 7*) and formed the following command: **sudo netwox 40 -l 10.0.2.15 -m 10.0.2.16 -o 34214 -p 23 -q 2298519912 -r 3792038262 -H "636174"**

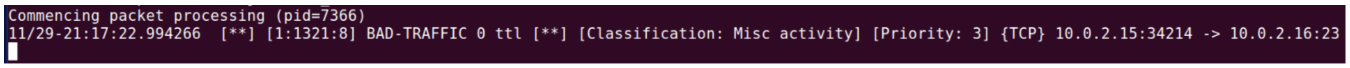


```
91 2020-11-29… 10.0.2.16   10.0.2.15   TELNET        134 Telnet Data ...
92 2020-11-29… 10.0.2.15   10.0.2.16   TCP            66 34214 → 23 [ACK] Seq=2298519912 Ack=3792038330 Win=30336 Len=

    [Stream index: 0]
    [TCP Segment Len: 68]
    Sequence number: 3792038262
    [Next sequence number: 3792038330]
    Acknowledgment number: 2298519912
    Header Length: 32 bytes
  ▸ Flags: 0x018 (PSH, ACK)
    Window size value: 227
```

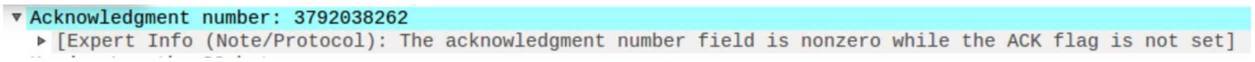
Image 7

    d. This resulted in an invalid packet. Using the feedback in wireshark and Snort, I found what I had done wrong (*Image 8 and Image 9*)



```
Commencing packet processing (pid=7366)
11/29-21:17:22.994266  [**] [1:1321:8] BAD-TRAFFIC 0 ttl [**] [Classification: Misc activity] [Priority: 3] {TCP} 10.0.2.15:34214 -> 10.0.2.16:23
```

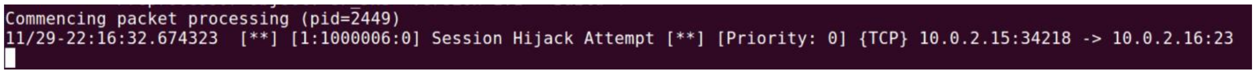
Image 8



```
▾ Acknowledgment number: 3792038262
  ▸ [Expert Info (Note/Protocol): The acknowledgment number field is nonzero while the ACK flag is not set]
```

Image 9

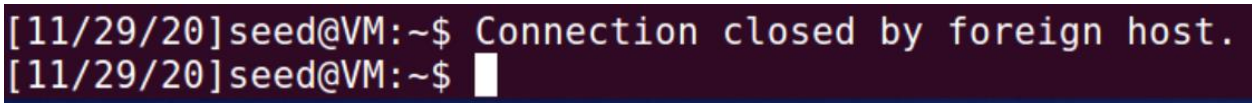
    e. I then restarted the connection and repeated step c. From this new packet and my earlier findings, I formed the following command: **sudo netwox 40 -g -j 64 -l 10.0.2.15 -m 10.0.2.16 -o 34218 -p 23 -q 1481560908 -r 3361055089 -z -H "636174"**
    f. After running the above command, my rule fired (Image 10)



```
Commencing packet processing (pid=2449)
11/29-22:16:32.674323  [**] [1:1000006:0] Session Hijack Attempt [**] [Priority: 0] {TCP} 10.0.2.15:34218 -> 10.0.2.16:23
```

Image 10

    g. I checked on the client's side to verify my rule responded to the attempted hijack by closing the connection (the react option in the rule) (Image 11)



```
[11/29/20]seed@VM:~$ Connection closed by foreign host.
[11/29/20]seed@VM:~$ 
```

Image 11

Note: I only showed one rule but as previously mentioned, modeled after the rules included with snort, the idea would be to create rules for various commands that could be dangerous in this sort of attack.

Conclusion

I found it really interesting to have played the red side in assignment 4 and then the blue side on assignment 5. It was a bit challenging to use the results of my previous assignment for this one as I had a few difficulties in assignment 4. Although, in the end, I believe expanding on the previous topic helped me fill in the gaps I had in my knowledge. It definitely put into perspective how it can be difficult to both attack and defend as both parties need to try and anticipate the other's actions/implementations and then engineer their next move accordingly.