Lab 04: Intrusion Detection

5/14/2025

Executive Summary

This lab was about getting hands-on experience with SNORT, a popular open-source Intrusion Detection System (IDS). I used a Windows-based virtual machine in the SimSpace cyber range to get everything up and running. SNORT lets security teams monitor network traffic and detect suspicious activity in real time, and the goal here was to walk through that entire process: setting it up, looking through its configuration files, testing traffic, and writing our own detection rules based on actual vulnerabilities (CVEs). It wasn't just about getting it to run,it was about understanding how it works and why it's set up the way it is.

One of the first things I realized while doing this lab is just how much detail goes into identifying the correct network interface and making sure SNORT is looking at the right traffic. If you're not paying attention to which interface is active or capturing data, SNORT is basically useless. That made the early steps of running snort.exe -W and validating the config feel more important than I expected. It's a reminder that even in cybersecurity, small missteps can create big blind spots.

The process of digging through the configuration file felt a bit like trying to understand someone else's train of thought. It's all there, rule paths, variable declarations, and which rule sets are being used, but it's easy to miss something if you're not looking closely. I started seeing how modular SNORT really is. You can include or exclude rule files based on your needs, and that level of control is something that would be super useful in a real-world environment.

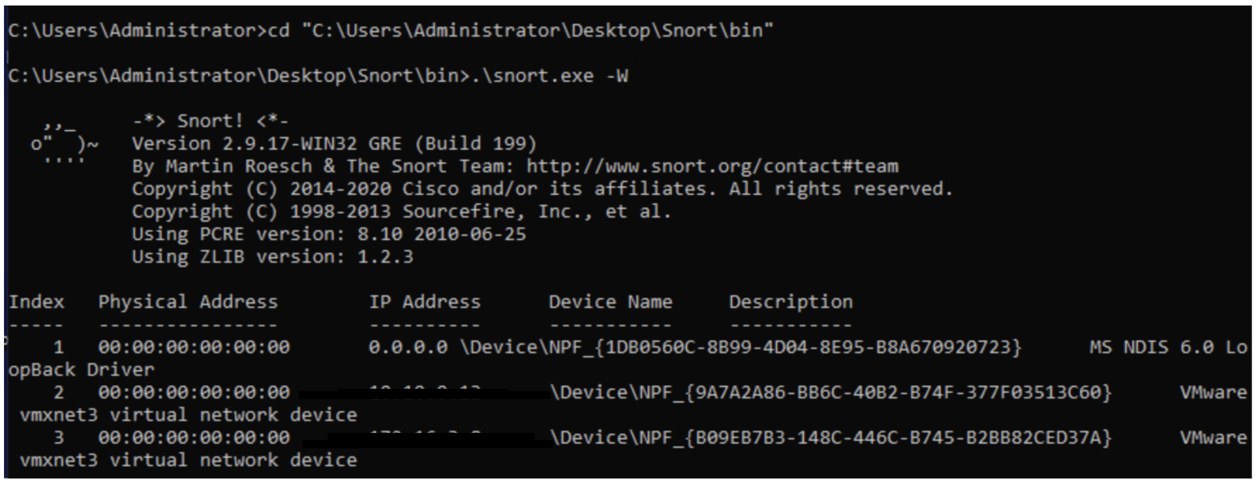
One of my favorite parts of this lab was working with both SNORT and Wireshark together. Wireshark gave me a visual map of the network activity, while SNORT was translating those packets into alerts. It felt like learning two languages and realizing they're describing the same thing. Seeing how SNORT responded to basic commands like curl or ping made me appreciate how powerful even simple traffic can be when viewed through the lens of security.

Overall, this lab helped me build more than just technical skills, it built confidence. Now I know how to set up SNORT, check its configuration, monitor traffic, and write custom detection logic. That's a big deal because it means I can bring real value to any future job or project that involves network security. The whole process made SNORT feel less like a mysterious black box and more like a tool I actually know how to use.

Breakpoint 1: Starting SNORT

Starting off this lab, my first goal was to get SNORT up and running on the virtual machine. Since SNORT works by monitoring network traffic, I had to make sure it was looking at the right network interface. If you don't get that part right, SNORT won't see any traffic at all, so this step was more important than I originally thought.

To figure that out, I opened the Command Prompt inside the SNORT directory and ran the command snort.exe -W. This showed me a list of all the network interfaces available on the system, each with a number next to it. After checking the IP addresses, I saw that interface number 2 matched the one used by my SNORT VM ([redacted]), so I chose that one to use.



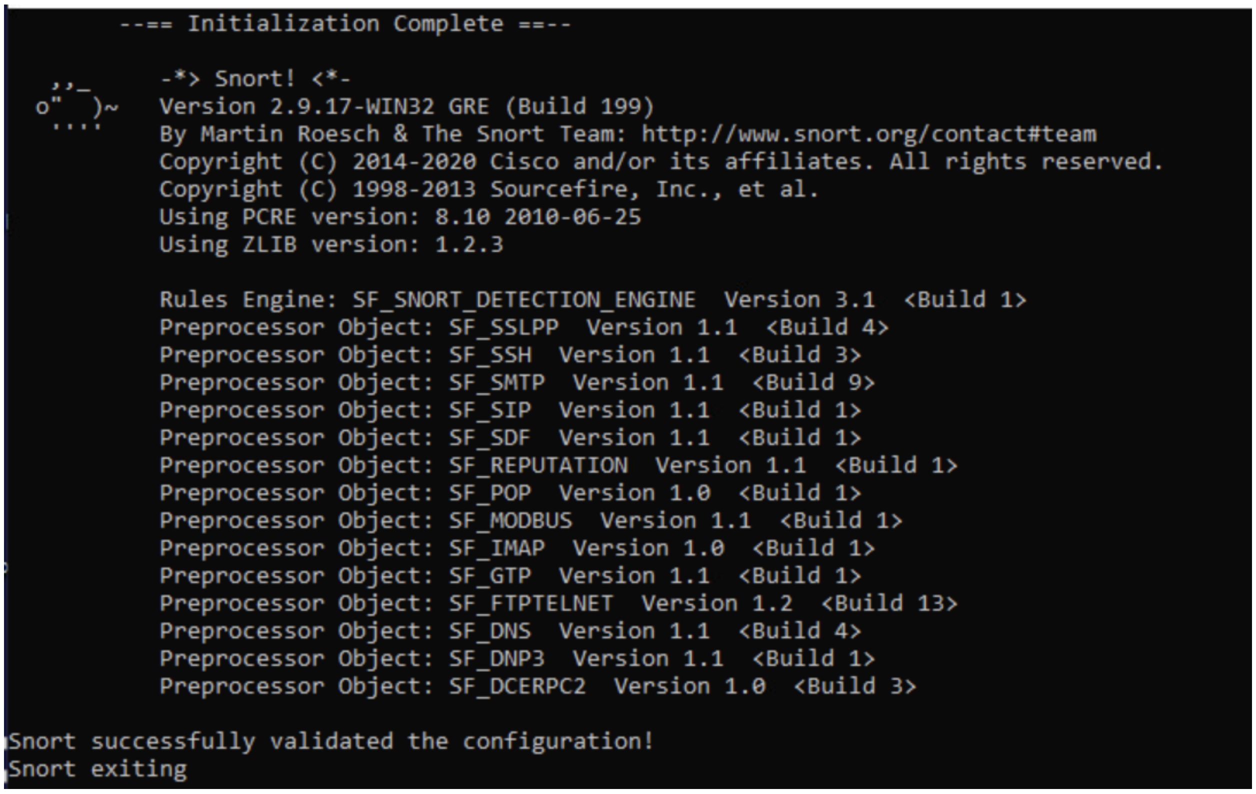
```
C:\Users\Administrator>cd "C:\Users\Administrator\Desktop\Snort\bin"

C:\Users\Administrator\Desktop\Snort\bin>.\snort.exe -W

   ,,_     -*> Snort! <*-
  o"  )~   Version 2.9.17-WIN32 GRE (Build 199)
   ''''    By Martin Roesch & The Snort Team: http://www.snort.org/contact#team
           Copyright (C) 2014-2020 Cisco and/or its affiliates. All rights reserved.
           Copyright (C) 1998-2013 Sourcefire, Inc., et al.
           Using PCRE version: 8.10 2010-06-25
           Using ZLIB version: 1.2.3

Index   Physical Address        IP Address      Device Name     Description
-----   ----------------        ----------      -----------     -----------
    1   00:00:00:00:00:00       0.0.0.0 \Device\NPF_{1DB0560C-8B99-4D04-8E95-B8A670920723}      MS NDIS 6.0 Lo
opBack Driver
    2   00:00:00:00:00:00 [redacted] \Device\NPF_{9A7A2A86-BB6C-40B2-B74F-377F03513C60}      VMware
 vmxnet3 virtual network device
    3   00:00:00:00:00:00 [redacted] \Device\NPF_{B09EB7B3-148C-446C-B745-B2BB82CED37A}      VMware
 vmxnet3 virtual network device
```

The list of interfaces from running snort.exe -W

Next, I had to make sure SNORT's configuration file didn't have any errors. I used this command to test it:

```
snort.exe -T -i 2 -c
"C:\Users\Administrator\Desktop\Snort\etc\snort.conf"
```

Running that command runs SNORT in test mode and checks to make sure all the rule paths and settings are correct. I waited a few seconds, and eventually, I got a message saying the configuration was successfully validated. That was a huge relief because I knew it meant SNORT wouldn't crash the second I tried to actually start it.



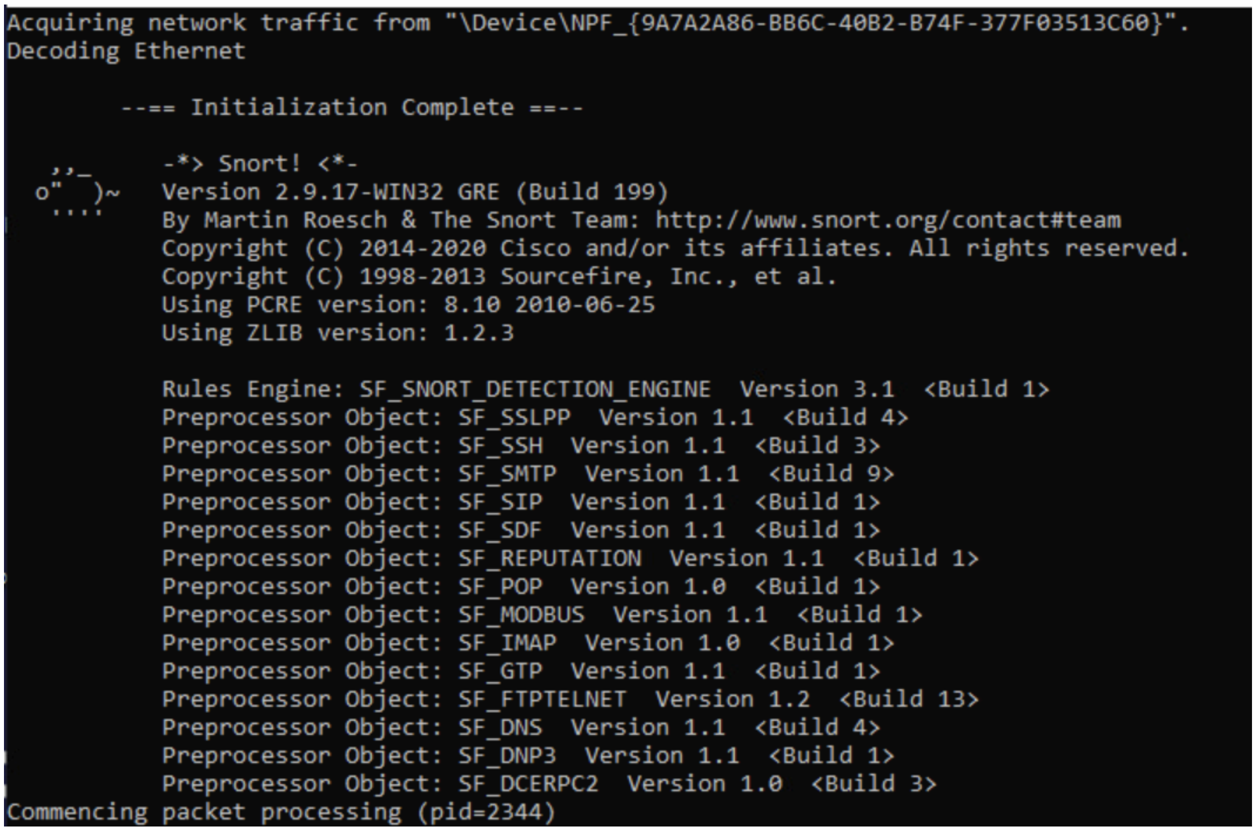
```
        --== Initialization Complete ==--

   ,,_     -*> Snort! <*-
  o"  )~   Version 2.9.17-WIN32 GRE (Build 199)
   ''''    By Martin Roesch & The Snort Team: http://www.snort.org/contact#team
           Copyright (C) 2014-2020 Cisco and/or its affiliates. All rights reserved.
           Copyright (C) 1998-2013 Sourcefire, Inc., et al.
           Using PCRE version: 8.10 2010-06-25
           Using ZLIB version: 1.2.3

           Rules Engine: SF_SNORT_DETECTION_ENGINE  Version 3.1  <Build 1>
           Preprocessor Object: SF_SSLPP  Version 1.1  <Build 4>
           Preprocessor Object: SF_SSH  Version 1.1  <Build 3>
           Preprocessor Object: SF_SMTP  Version 1.1  <Build 9>
           Preprocessor Object: SF_SIP  Version 1.1  <Build 1>
           Preprocessor Object: SF_SDF  Version 1.1  <Build 1>
           Preprocessor Object: SF_REPUTATION  Version 1.1  <Build 1>
           Preprocessor Object: SF_POP  Version 1.0  <Build 1>
           Preprocessor Object: SF_MODBUS  Version 1.1  <Build 1>
           Preprocessor Object: SF_IMAP  Version 1.0  <Build 1>
           Preprocessor Object: SF_GTP  Version 1.1  <Build 1>
           Preprocessor Object: SF_FTPTELNET  Version 1.2  <Build 13>
           Preprocessor Object: SF_DNS  Version 1.1  <Build 4>
           Preprocessor Object: SF_DNP3  Version 1.1  <Build 1>
           Preprocessor Object: SF_DCERPC2  Version 1.0  <Build 3>

Snort successfully validated the configuration!
Snort exiting
```

Validation success message from SNORT

After that, it was time to start SNORT in live mode so it could start analyzing real traffic. I ran this command:

```
snort.exe -A console -i 2 -c
"C:\Users\Administrator\Desktop\Snort\etc\snort.conf"
```

This started SNORT and began showing packet data in the terminal. It was kind of cool to see that in real time, it felt like I had just turned on a live feed of what was happening on the network. I could actually see that SNORT was working and watching for anything suspicious.



```
Acquiring network traffic from "\Device\NPF_{9A7A2A86-BB6C-40B2-B74F-377F03513C60}".
Decoding Ethernet

        --== Initialization Complete ==--

   ,,_     -*> Snort! <*-
  o"  )~   Version 2.9.17-WIN32 GRE (Build 199)
   ''''    By Martin Roesch & The Snort Team: http://www.snort.org/contact#team
           Copyright (C) 2014-2020 Cisco and/or its affiliates. All rights reserved.
           Copyright (C) 1998-2013 Sourcefire, Inc., et al.
           Using PCRE version: 8.10 2010-06-25
           Using ZLIB version: 1.2.3

           Rules Engine: SF_SNORT_DETECTION_ENGINE  Version 3.1  <Build 1>
           Preprocessor Object: SF_SSLPP  Version 1.1  <Build 4>
           Preprocessor Object: SF_SSH  Version 1.1  <Build 3>
           Preprocessor Object: SF_SMTP  Version 1.1  <Build 9>
           Preprocessor Object: SF_SIP  Version 1.1  <Build 1>
           Preprocessor Object: SF_SDF  Version 1.1  <Build 1>
           Preprocessor Object: SF_REPUTATION  Version 1.1  <Build 1>
           Preprocessor Object: SF_POP  Version 1.0  <Build 1>
           Preprocessor Object: SF_MODBUS  Version 1.1  <Build 1>
           Preprocessor Object: SF_IMAP  Version 1.0  <Build 1>
           Preprocessor Object: SF_GTP  Version 1.1  <Build 1>
           Preprocessor Object: SF_FTPTELNET  Version 1.2  <Build 13>
           Preprocessor Object: SF_DNS  Version 1.1  <Build 4>
           Preprocessor Object: SF_DNP3  Version 1.1  <Build 1>
           Preprocessor Object: SF_DCERPC2  Version 1.0  <Build 3>
Commencing packet processing (pid=2344)
```

SNORT live mode showing "Commencing packet processing"

Lastly, even though this lab only had us run SNORT in IDS mode (which means it just watches and alerts), I also learned that SNORT can run as an IPS too. That means it can actively block traffic instead of just logging it. To do that, you'd just add this line to the config file:

```
config policy_mode:inline
```

We didn't go that far in this lab, but it's helpful to know how easily SNORT can be configured to either passively detect threats or actively stop them, depending on what the situation calls for.

One thing that really stuck with me during this part of the lab was how even small mistakes, like choosing the wrong interface or missing a typo in the config—could totally break the setup. It made me realize how detail-oriented network security work really is. SNORT is powerful, but it’s not forgiving. You have to be deliberate about every step, which honestly made me appreciate the process more. When the validation passed and SNORT started running without issues, it felt like a real win.

I also started thinking about how this kind of setup would translate outside the lab. In a real network environment, you’d probably be dealing with way more interfaces, and maybe even multiple sensors. Knowing how to pinpoint the right interface and validate that your IDS is working correctly would be crucial, especially if you're responsible for monitoring traffic across different segments or detecting attacks in real time. This lab helped me build that baseline confidence in navigating those tools.

Lastly, it was kind of surprising to see how much information SNORT gives you right out of the gate. Just by running it with the default rules, I could see it picking up on basic traffic patterns and logging activity. I started to realize how SNORT isn’t just about big flashy threats, it’s also useful for picking up on the small, weird behaviors that could be early warning signs. Even though this was just the setup phase, it laid the groundwork for everything else I’d do in the lab.

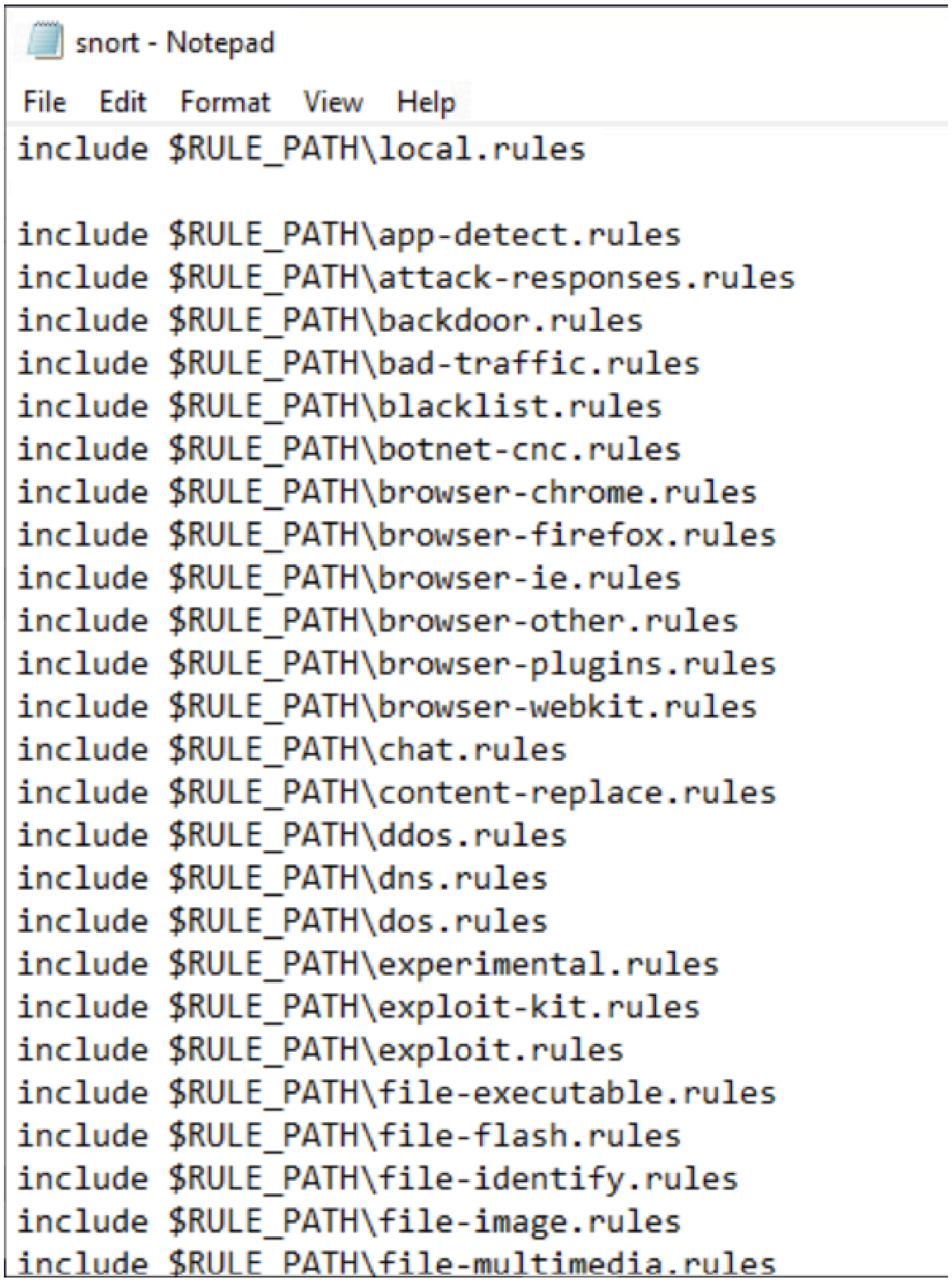
Breakpoint 2: SNORT Configuration Analysis

After getting SNORT up and running, the next part of the lab had me diving into the configuration settings to understand how everything was set up. At first glance, the snort.conf file looked pretty intimidating, it's long, full of variables, includes, and syntax that doesn't immediately make sense if you haven't worked with it before. But once I started going through it line by line, it began to click. It's actually laid out in a logical way, and once you understand the role of each section, it becomes a lot easier to navigate.

One of the first things I looked for was the variable that tells SNORT where to find its rule files. I found the line:

```
var RULE_PATH ../rules
```

This line basically sets a shortcut so SNORT knows to look in the "rules" folder for all the .rules files that define what to look for in the traffic. This one line is used throughout the config file so that SNORT can include rules from different files without typing out the full path every time. It's a small thing, but it's a smart way to make the file easier to maintain.



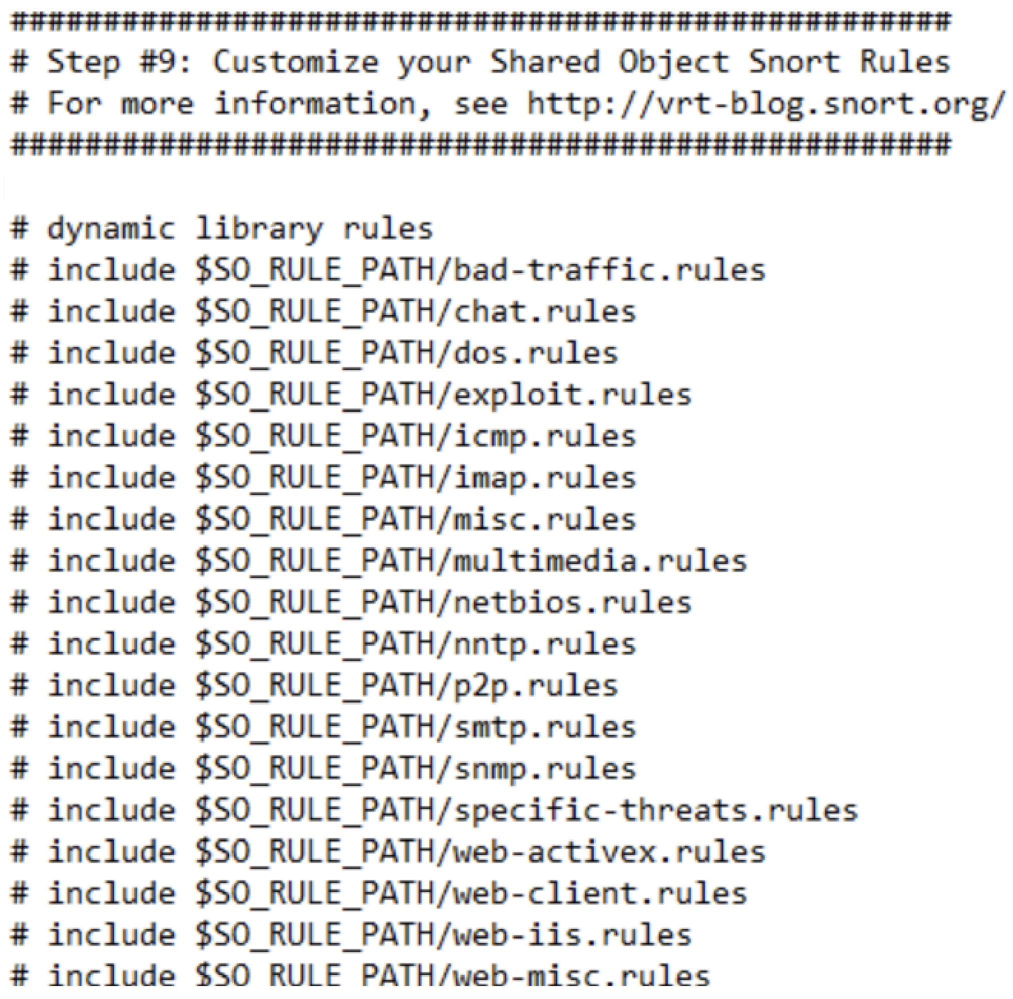
```
include $RULE_PATH\local.rules

include $RULE_PATH\app-detect.rules
include $RULE_PATH\attack-responses.rules
include $RULE_PATH\backdoor.rules
include $RULE_PATH\bad-traffic.rules
include $RULE_PATH\blacklist.rules
include $RULE_PATH\botnet-cnc.rules
include $RULE_PATH\browser-chrome.rules
include $RULE_PATH\browser-firefox.rules
include $RULE_PATH\browser-ie.rules
include $RULE_PATH\browser-other.rules
include $RULE_PATH\browser-plugins.rules
include $RULE_PATH\browser-webkit.rules
include $RULE_PATH\chat.rules
include $RULE_PATH\content-replace.rules
include $RULE_PATH\ddos.rules
include $RULE_PATH\dns.rules
include $RULE_PATH\dos.rules
include $RULE_PATH\experimental.rules
include $RULE_PATH\exploit-kit.rules
include $RULE_PATH\exploit.rules
include $RULE_PATH\file-executable.rules
include $RULE_PATH\file-flash.rules
include $RULE_PATH\file-identify.rules
include $RULE_PATH\file-image.rules
include $RULE_PATH\file-multimedia.rules
```

Line in snort.conf showing var RULE_PATH ../rules

Next, I found all the included statements. These are what actually tell SNORT which specific rule files to load. Some of the ones I saw included:

```
include $RULE_PATH/local.rules
include $RULE_PATH/bad-traffic.rules

include $RULE_PATH/exploit.rules
```

Each of these is like a category of rules. The “local.rules” file is where custom rules go, so that’s where I’ll be putting the ones I write later in the lab. The others are prewritten rule sets that look for different kinds of suspicious activity. It was helpful to see how modular this system is. You can turn off or comment out rule sets you don’t need, or add your own as needed.



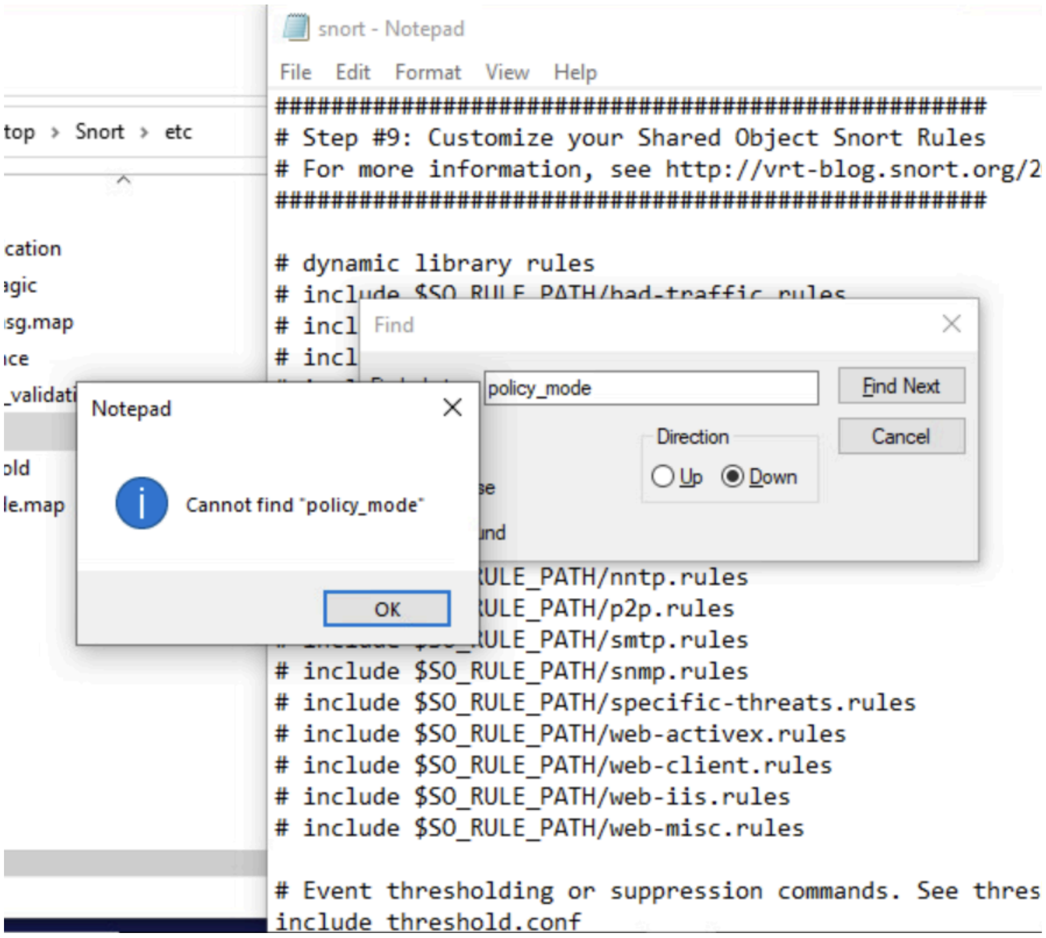
```
###################################################
# Step #9: Customize your Shared Object Snort Rules
# For more information, see http://vrt-blog.snort.org/
###################################################

# dynamic library rules
# include $SO_RULE_PATH/bad-traffic.rules
# include $SO_RULE_PATH/chat.rules
# include $SO_RULE_PATH/dos.rules
# include $SO_RULE_PATH/exploit.rules
# include $SO_RULE_PATH/icmp.rules
# include $SO_RULE_PATH/imap.rules
# include $SO_RULE_PATH/misc.rules
# include $SO_RULE_PATH/multimedia.rules
# include $SO_RULE_PATH/netbios.rules
# include $SO_RULE_PATH/nntp.rules
# include $SO_RULE_PATH/p2p.rules
# include $SO_RULE_PATH/smtp.rules
# include $SO_RULE_PATH/snmp.rules
# include $SO_RULE_PATH/specific-threats.rules
# include $SO_RULE_PATH/web-activex.rules
# include $SO_RULE_PATH/web-client.rules
# include $SO_RULE_PATH/web-iis.rules
# include $SO_RULE_PATH/web-misc.rules
```

include lines showing rule file categories

To figure out if SNORT was set to run as an IDS or an IPS, I used the search feature in Notepad to look for the word policy_mode. I didn't find anything, which told me SNORT is in its default IDS mode. If it had been set up for inline IPS mode, it would've had a line like:

```
config policy_mode:inline
```

Since it wasn't there, that confirmed we're working with SNORT in a passive detection role, it's observing traffic but not blocking anything.



```
#######################################################
# Step #9: Customize your Shared Object Snort Rules
# For more information, see http://vrt-blog.snort.org/2
#######################################################

# dynamic library rules
# include $SO_RULE_PATH/bad-traffic.rules
# incl
# incl
ULE_PATH/nntp.rules
ULE_PATH/p2p.rules
ULE_PATH/smtp.rules
# include $SO_RULE_PATH/snmp.rules
# include $SO_RULE_PATH/specific-threats.rules
# include $SO_RULE_PATH/web-activex.rules
# include $SO_RULE_PATH/web-client.rules
# include $SO_RULE_PATH/web-iis.rules
# include $SO_RULE_PATH/web-misc.rules

# Event thresholding or suppression commands. See thres
include threshold.conf
```

policy_mode

Cannot find "policy_mode"

Search result in Notepad showing "policy_mode" not found

If I ever wanted to switch it into IPS mode, I'd just need to add that line to the config and make sure it's properly positioned. I'd also probably need to do more setup, like running SNORT on a system that supports inline packet handling (like a Linux bridge or with NFQUEUE). But just seeing how easy it is to flip the setting shows how flexible SNORT really is.

What stood out to me while reading through the config file was how customizable everything is. I could define new paths, add or remove rule sets, change network variables, and pretty much shape SNORT into exactly what I need. In a way, it felt like putting together a toolkit where every piece has a purpose. And because it's just a text file, you don't need special software or licenses to change it, just a basic understanding of how it all connects.

This part of the lab taught me that configuring SNORT isn't just about getting it to work. It's about tailoring it to your environment. Whether that's a small internal network or a large-scale enterprise setup, knowing how to read and modify the snort.conf file gives you control over how SNORT behaves. By the end of this section, I felt way more confident in my ability to make meaningful changes and troubleshoot issues based on what I saw in the config.

Another thing I took away from this was just how important it is to keep your rule sets organized. When you start dealing with dozens of rules or more, having everything grouped by type, like bad traffic, exploits, or user-defined, isn't just convenient, it's necessary. I could see how in a larger organization, maintaining these files would be part of someone's regular workflow to make sure detection stays accurate and up to date.

It also made me think about how changes to a config file like this could impact detection in subtle ways. If you comment out the wrong rule file or forget to define a variable, SNORT won't necessarily break, it might just silently skip over certain detections. That idea pushed me to double-check my work and be more intentional about the edits I made. It gave me a new appreciation for why documentation and version control matter even with something as “simple” as a config file.

Overall, going through the SNORT configuration was a reminder that cybersecurity isn’t just about flashy exploits or attacks, it’s about understanding the systems and tools well enough to make them work for you. Being able to adjust how SNORT operates at a foundational level felt empowering, and it gave me a new level of respect for people who build and maintain IDS/IPS systems in the real world.

Breakpoint 3: Capturing and Analyzing network Traffic

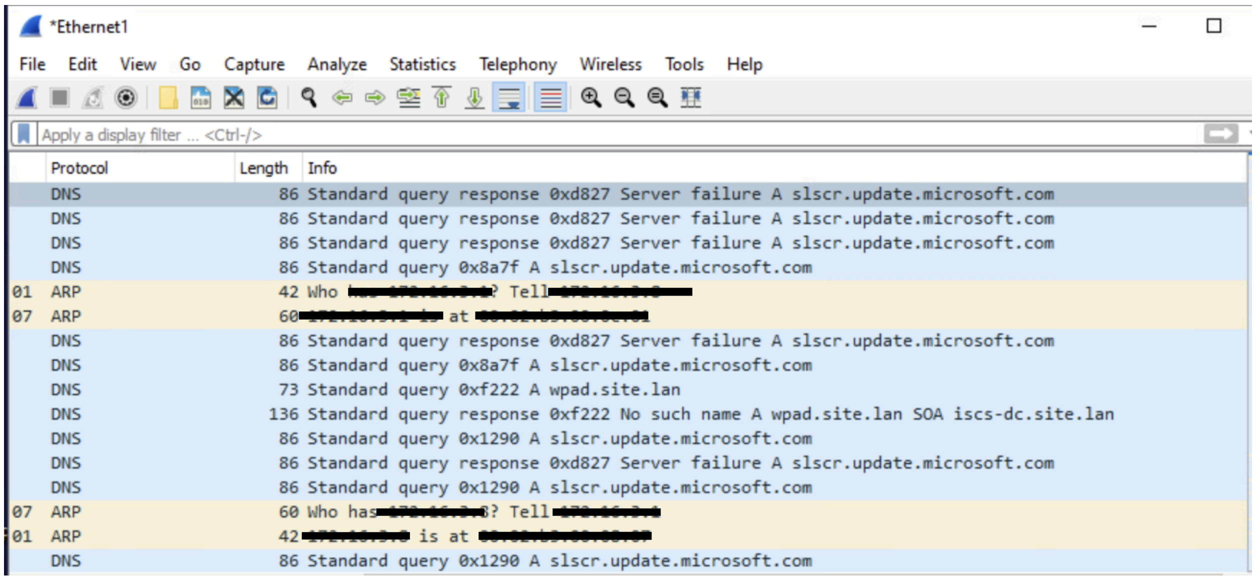
After getting SNORT set up and reading through its configuration, I got to actually generate some traffic and watch how SNORT handled it. The idea here was to simulate different types of network activity using a second VM, basically playing both sides: the defender and the “attacker.” It made the whole thing feel real and gave me a better sense of how an IDS functions in practice.

I started by launching a second virtual machine on the same network as the SNORT VM. From there, I ran a few basic commands to create network traffic aimed at the SNORT machine. Some of the commands I used were:

```
ping [redacted]            # ICMP
ftp [redacted]             # FTP
curl http://[redacted]     # HTTP
```

Each of these represents a different type of protocol, and even though they're simple, they're common enough that they show up in real-world networks all the time. I wasn't trying to exploit anything here, just trying to see if SNORT would notice the activity, and it definitely did.

To get a visual look at the network activity, I opened Wireshark on the SNORT VM and started capturing on interface Ethernet1, which was the same one SNORT was monitoring. Right away, I could see packets flying across the screen. I saw DNS queries, ARP traffic, and some of the traffic I had just generated from the second VM. It was satisfying to see proof that the network was active and that my traffic was being picked up correctly.



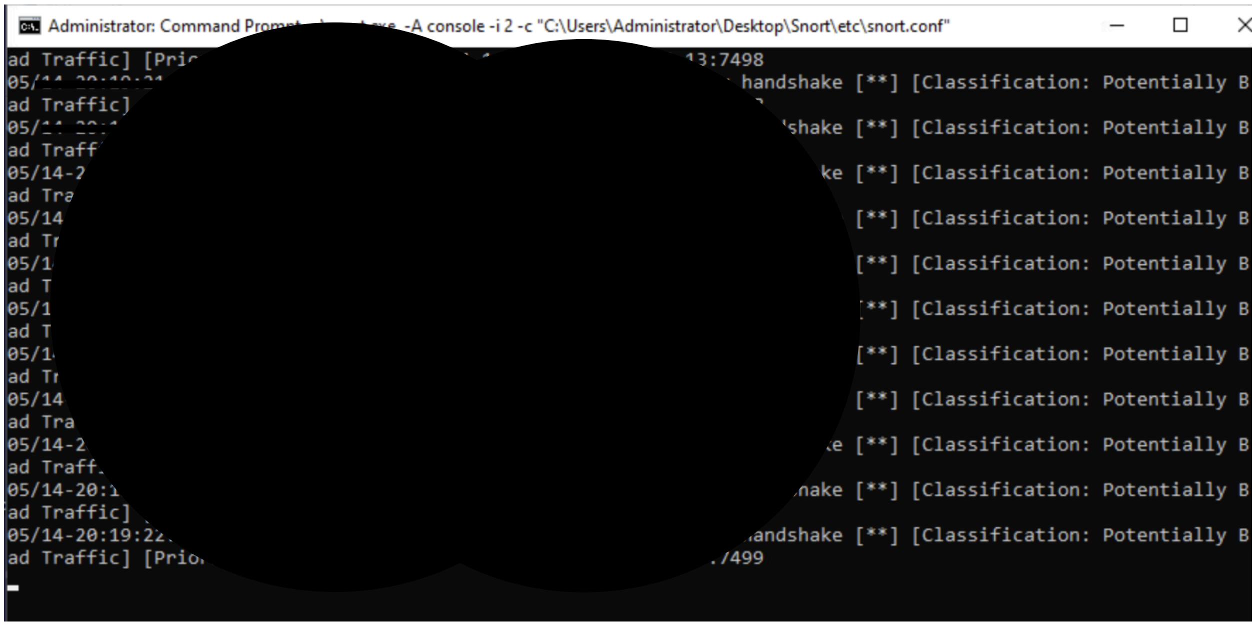
```
     Protocol                Length Info
     DNS                         86 Standard query response 0xd827 Server failure A slscr.update.microsoft.com
     DNS                         86 Standard query response 0xd827 Server failure A slscr.update.microsoft.com
     DNS                         86 Standard query response 0xd827 Server failure A slscr.update.microsoft.com
     DNS                         86 Standard query 0x8a7f A slscr.update.microsoft.com
01   ARP                         42 Who [redacted]? Tell[redacted]
07   ARP                         60[redacted] at [redacted]
     DNS                         86 Standard query response 0xd827 Server failure A slscr.update.microsoft.com
     DNS                         86 Standard query 0x8a7f A slscr.update.microsoft.com
     DNS                         73 Standard query 0xf222 A wpad.site.lan
     DNS                        136 Standard query response 0xf222 No such name A wpad.site.lan SOA iscs-dc.site.lan
     DNS                         86 Standard query 0x1290 A slscr.update.microsoft.com
     DNS                         86 Standard query response 0xd827 Server failure A slscr.update.microsoft.com
     DNS                         86 Standard query 0x1290 A slscr.update.microsoft.com
07   ARP                         60 Who has[redacted]? Tell[redacted]
01   ARP                         42[redacted] is at [redacted]
     DNS                         86 Standard query 0x1290 A slscr.update.microsoft.com
```

Wireshark capture showing DNS, ARP, and test traffic

The even cooler part was watching SNORT respond. In the terminal window where SNORT was running, I started seeing alerts pop up. One of them was for a TCP connection that didn't complete a full handshake, which SNORT flagged as potentially suspicious. This kind of thing can happen during port scans, which made sense since I was connecting to ports without fully logging in.

[**] [129:20:1] TCP session without 3-way handshake [**]

[Priority: 2] {TCP} [redacted]:8140 -> [redacted]:7499



```
Administrator: Command Prom[redacted]-A console -i 2 -c "C:\Users\Administrator\Desktop\Snort\etc\snort.conf"
ad Traffic] [Pri                                    13:7498
05/                                                handshake [**] [Classification: Potentially B
ad Traffic]
05/                                                 shake [**] [Classification: Potentially B
ad Traff
05/14-2                                               ke [**] [Classification: Potentially B
ad Tra
05/14                                                    [**] [Classification: Potentially B
ad Tr
05/1                                                     [**] [Classification: Potentially B
ad T
05/1                                                      **] [Classification: Potentially B
ad T
05/1                                                     [**] [Classification: Potentially B
ad Tr
05/14                                                    [**] [Classification: Potentially B
ad Tra
05/14-2                                                e [**] [Classification: Potentially B
ad Traff
05/14-20:1                                           nake [**] [Classification: Potentially B
ad Traffic]
05/14-20:19:2                                       andshake [**] [Classification: Potentially B
ad Traffic] [Prio                                   /499
```

SNORT alert showing TCP session without 3-way handshake

That moment really made the whole setup feel legit. SNORT wasn't just passively sitting there, it was actively watching and interpreting what was going on. It didn't just collect data; it applied logic and raised flags when something looked off. That's the difference between a packet sniffer and an intrusion detection system, and this part of the lab really brought that home.

From a Blue Team perspective, the traffic I generated looked innocent on the surface, just a few pings, a curl command, maybe an FTP login, but those same actions could be used by an attacker to map a network or test for open ports. Seeing how SNORT classified these actions helped me understand how important context is when analyzing alerts. A single ping isn't a threat, but a pattern of them could indicate reconnaissance. That's the kind of judgment call that analysts have to make all the time.

One thing I realized is that even basic network tools, like ping, FTP, or HTTP requests, can be valuable for both attackers and defenders. In this case, I was using them for a controlled lab test, but in a real environment, the same tools could be used maliciously. Being able to see that activity in Wireshark, and more importantly, knowing that SNORT flagged some of it, gave me a lot more confidence in how detection systems work behind the scenes.

Overall, this part of the lab felt like the turning point where everything I had set up before finally came together. It wasn't just theory anymore, I was generating traffic, capturing it, and watching SNORT detect it in real time. That hands-on aspect was incredibly helpful, and it gave me a new appreciation for what goes into monitoring a network for threats. I left this section feeling like I had a better understanding of both the technical and the strategic side of traffic analysis.

Breakpoint 4: Writing Custom SNORT Rules

After spending time observing how SNORT detects predefined threats using existing rule sets, I got to write my own detection rules based on real-world CVEs. It felt like I was stepping into the shoes of a security analyst or SOC engineer, crafting logic to catch known vulnerabilities before they can be exploited. And while the syntax was a little tricky at first, once I understood the structure, it started to feel kind of natural.

SNORT rules follow a consistent pattern that makes them both readable and powerful. They start with an action (alert, log, etc.), followed by the protocol, source and destination IPs and ports, and then the options inside parentheses. For example:

```
alert tcp any any -> any 80 (msg:"Example"; content:"bad stuff";
sid:1000001; rev:1;)
```

This basic structure forms the skeleton of every rule. The part inside the parentheses is where you define what SNORT should be looking for in the packet. That's where the real magic happens.

The first CVE I chose was CVE-2021-44228, also known as Log4 Shell. It was a major vulnerability in the Log4j library that allowed attackers to perform remote code execution through malicious input strings. I wrote a rule that looks for part of the JNDI string typically used in these attacks:

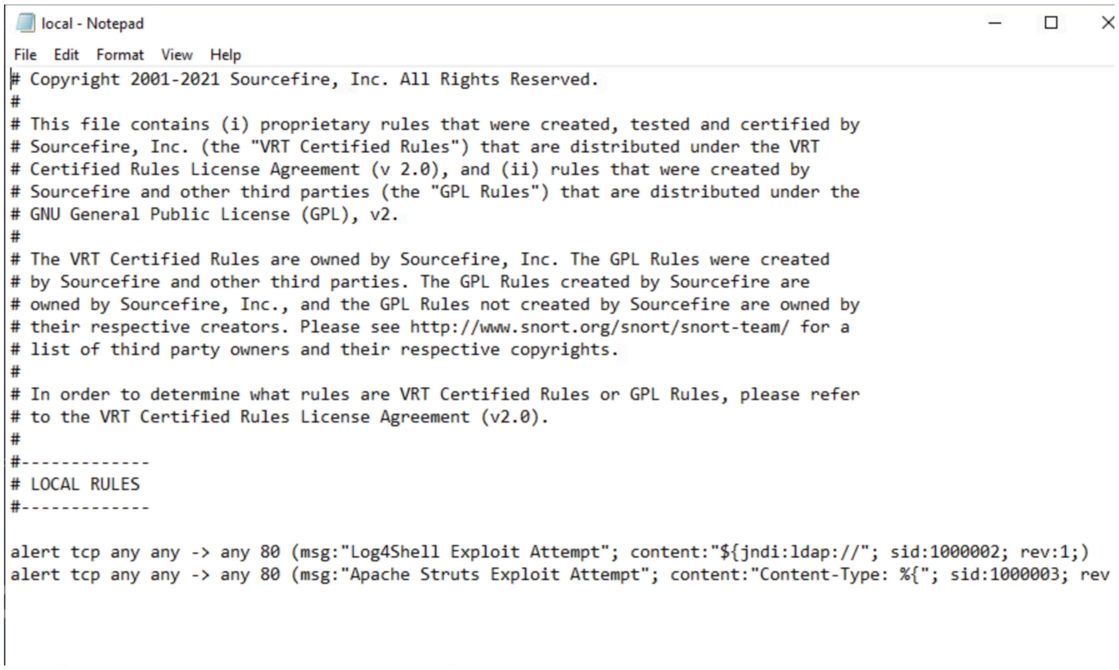
```
alert tcp any any -> any 80 (msg:"Log4Shell Exploit Attempt";
content:"${jndi:ldap://"; sid:1000002; rev:1;)
```

This rule watches for any TCP traffic going to port 80 that contains the string ${jndi:ldap://, which is a strong indicator of a Log4 Shell exploit attempt. It's a simple rule, but it's targeted and effective. I used the content option to match the string ${jndi:ldap://, which is a unique signature of the attack. It's commonly used in HTTP headers and user input fields that the Log4j logger might process. Since most of these exploits target web servers, I scoped the rule to TCP traffic on port 80. There's a small chance of a false positive if someone uses this string in a non-malicious context, but that's rare in most networks.

The second rule I created was based on CVE-2017-5638, which affected Apache Struts and also allowed remote code execution. The vulnerability could be triggered by sending a specially crafted Content-Type header. I made a rule to catch that by looking for the %{ pattern, which is part of the exploit:

```
alert tcp any any -> any 80 (msg:"Apache Struts Exploit
Attempt"; content:"Content-Type: %{"; sid:1000003; rev:1;)
```

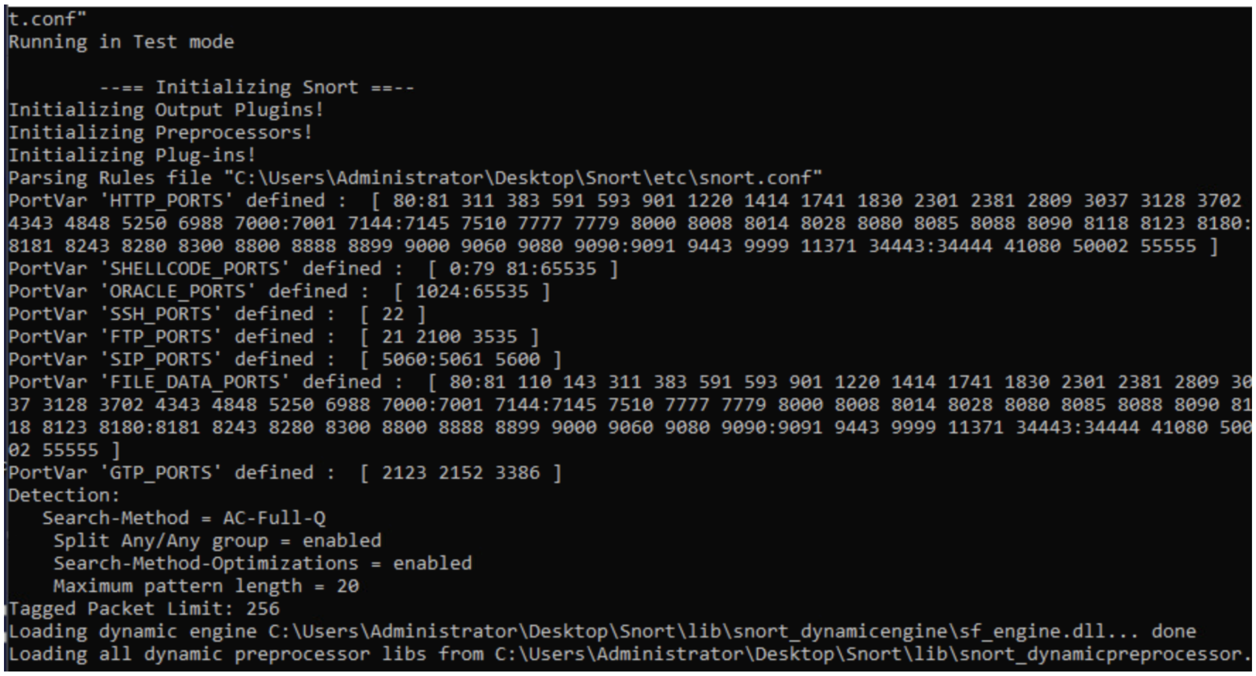
Like the first one, this rule scans HTTP traffic for a specific pattern that matches a known attack signature. I chose these two CVEs because they're both high-impact and well-documented, so it made sense to try and build rules that could detect them. I focused on detecting %{ inside the Content-Type header. This pattern triggers the OGNL parser, which is how attackers exploit the vulnerability. I could make this rule even more precise by checking for Content-Type: and %{ within a limited byte range using the within keyword, but even in its basic form, it should catch most exploit attempts. It's important to tune rules like this carefully to avoid flooding the logs with irrelevant alerts.



```
# Copyright 2001-2021 Sourcefire, Inc. All Rights Reserved.
#
# This file contains (i) proprietary rules that were created, tested and certified by
# Sourcefire, Inc. (the "VRT Certified Rules") that are distributed under the VRT
# Certified Rules License Agreement (v 2.0), and (ii) rules that were created by
# Sourcefire and other third parties (the "GPL Rules") that are distributed under the
# GNU General Public License (GPL), v2.
#
# The VRT Certified Rules are owned by Sourcefire, Inc. The GPL Rules were created
# by Sourcefire and other third parties. The GPL Rules created by Sourcefire are
# owned by Sourcefire, Inc., and the GPL Rules not created by Sourcefire are owned by
# their respective creators. Please see http://www.snort.org/snort/snort-team/ for a
# list of third party owners and their respective copyrights.
#
# In order to determine what rules are VRT Certified Rules or GPL Rules, please refer
# to the VRT Certified Rules License Agreement (v2.0).
#
#-------------
# LOCAL RULES
#-------------

alert tcp any any -> any 80 (msg:"Log4Shell Exploit Attempt"; content:"${jndi:ldap://"; sid:1000002; rev:1;)
alert tcp any any -> any 80 (msg:"Apache Struts Exploit Attempt"; content:"Content-Type: %{"; sid:1000003; rev
```

local.rules file with both custom rules added

Once I added both rules to the local.rules file, I ran SNORT in test mode again to make sure everything was working. I used the same -T command as before, and this time it also checked my custom rules. Seeing SNORT respond with “Configuration validated successfully” was super satisfying because it meant my rules were written correctly and ready to use.


```
t.conf"
Running in Test mode

        --== Initializing Snort ==--
Initializing Output Plugins!
Initializing Preprocessors!
Initializing Plug-ins!
Parsing Rules file "C:\Users\Administrator\Desktop\Snort\etc\snort.conf"
PortVar 'HTTP_PORTS' defined :  [ 80:81 311 383 591 593 901 1220 1414 1741 1830 2301 2381 2809 3037 3128 3702
4343 4848 5250 6988 7000:7001 7144:7145 7510 7777 7779 8000 8008 8014 8028 8080 8085 8088 8090 8118 8123 8180:
8181 8243 8280 8300 8800 8888 8899 9000 9060 9080 9090:9091 9443 9999 11371 34443:34444 41080 50002 55555 ]
PortVar 'SHELLCODE_PORTS' defined :  [ 0:79 81:65535 ]
PortVar 'ORACLE_PORTS' defined :  [ 1024:65535 ]
PortVar 'SSH_PORTS' defined :  [ 22 ]
PortVar 'FTP_PORTS' defined :  [ 21 2100 3535 ]
PortVar 'SIP_PORTS' defined :  [ 5060:5061 5600 ]
PortVar 'FILE_DATA_PORTS' defined :  [ 80:81 110 143 311 383 591 593 901 1220 1414 1741 1830 2301 2381 2809 30
37 3128 3702 4343 4848 5250 6988 7000:7001 7144:7145 7510 7777 7779 8000 8008 8014 8028 8080 8085 8088 8090 81
18 8123 8180:8181 8243 8280 8300 8800 8888 8899 9000 9060 9080 9090:9091 9443 9999 11371 34443:34444 41080 500
02 55555 ]
PortVar 'GTP_PORTS' defined :  [ 2123 2152 3386 ]
Detection:
   Search-Method = AC-Full-Q
    Split Any/Any group = enabled
    Search-Method-Optimizations = enabled
    Maximum pattern length = 20
Tagged Packet Limit: 256
Loading dynamic engine C:\Users\Administrator\Desktop\Snort\lib\snort_dynamicengine\sf_engine.dll... done
Loading all dynamic preprocessor libs from C:\Users\Administrator\Desktop\Snort\lib\snort_dynamicpreprocessor.
```
SNORT test mode showing successful validation of custom rules

Writing these rules helped me understand how security tools translate threat intelligence into something actionable. It's one thing to read about a vulnerability online, but it’s another thing entirely to take that knowledge and turn it into a working rule that a system like SNORT can actually use to catch attacks. It felt empowering to go from reading about exploits to creating defenses against them.

Another thing I noticed while writing these rules is how important precision is. You want to make your rule broad enough to catch the malicious traffic, but not so broad that it starts generating false positives. It’s a fine line, and it gave me a new appreciation for the people who write and maintain rule sets for enterprise security systems. Every rule is a balance between detection and noise.

Overall, this part of the lab tied everything together. I went from learning how SNORT works, to seeing it monitor traffic, and finally to customizing it with my own logic. It showed me that intrusion detection isn’t just about plugging in a tool and letting it run, it’s about knowing how to tune it for your environment, keep it updated with the latest threat intelligence, and adapt it to what’s happening on your network. That’s something I’ll definitely carry forward into any future security work I do.

Conclusion:

Looking back on this lab, I can honestly say it was one of the most valuable hands-on experiences I’ve had so far in cybersecurity. I didn’t just learn how to run SNORT, I learned how to troubleshoot it, configure it, interpret its output, and even extend it with my own rules. Each part of the lab built on the last, and by the time I reached the end, I had gone from simply trying to make SNORT run, to actually using it as a detection tool with real-world relevance.

One of the biggest takeaways for me was how important it is to understand the details, things like choosing the right interface, validating config files, or structuring a SNORT rule correctly. None of these steps are flashy, but if you skip them or get them wrong, the entire setup can fall apart. This lab made me more careful, more observant, and honestly, more patient.

What made this experience really click for me was how it connected the technical side of IDS with the mindset of a defender. It’s not just about knowing what buttons to press or what commands to run, it’s about understanding what’s happening under the hood and why. From sniffing packets to catching simulated threats, I got a taste of what it's like to be on the front lines of network security.

This lab didn’t just check boxes for a grade. It gave me practical skills that I can actually take with me, whether that’s in another class, a certification, or one day in a real job. I know how to install, run, and customize SNORT. I know how to use Wireshark to confirm traffic. I know how to write a rule that detects something dangerous. Most importantly, I know how to approach security problems with a mindset that combines curiosity, logic, and attention to detail.