Student’s name: Matheus Henrique Schaly

Professor’s name: Carlisle Adams

Course: CSI 4139 Design of Secure Computer Systems

Due date: 4th November, 2019

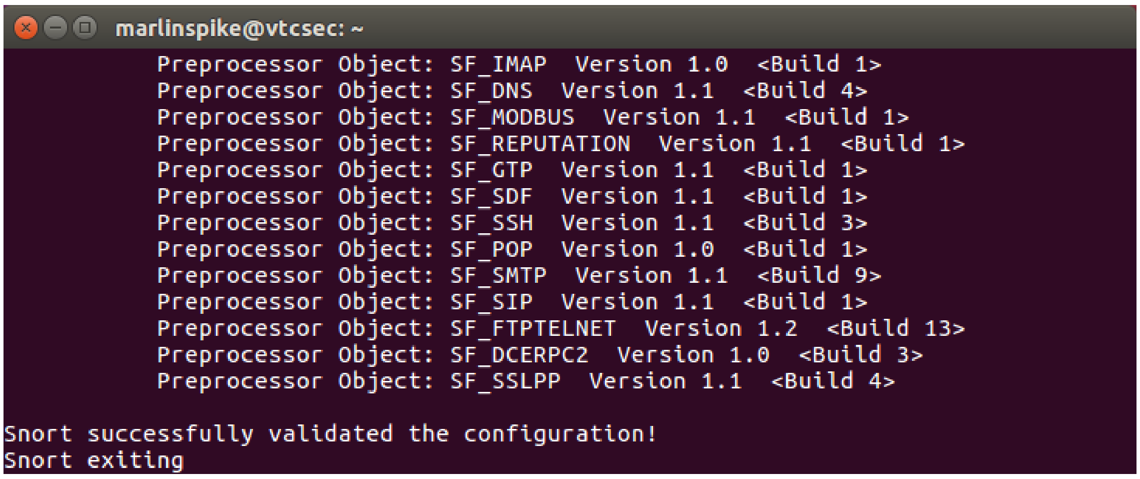
Report #3

Part A

We used the open-source Snort intrusion detection system. Firstly, we created a Ubunto VM and installed Snort. This is the machine is going to be sniffed by a second machine. The second machine is a Kali Linux VM, which is the attacker. We used the command *ifconfig* to get the Ubunto’s IP address as well as Kali Linux IP address. Then, we opened the main configuration file of Snort by using *sudo gedit /etc/snort/snort.conf*. After that, to get access to the rules, we set a var containing the path to the rules *var RULE_PATH etc/snort/rules*. Inside the *snort.conf* file, we can find several rules already defined, many of which are commented. Then, we opened a new terminal and used the command *sudo gedit /etc/snort/rules/ftp.rules* to open the rules associated with FTP. For example, the first rule was:

alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:”FTP MDTM overflow attempt”; flow:to_server,established; content:”MDTM”; nocase; isdattat:100, relative; pcre:”/^MDTM\s[^\n]{100}/smi”; reference:bugtraq,9751; reference:cve,2001-1021; reference:cve,2004-0330; reference:nessus,12080; classtype:attempted-admin; sid:2546; rev:5;)

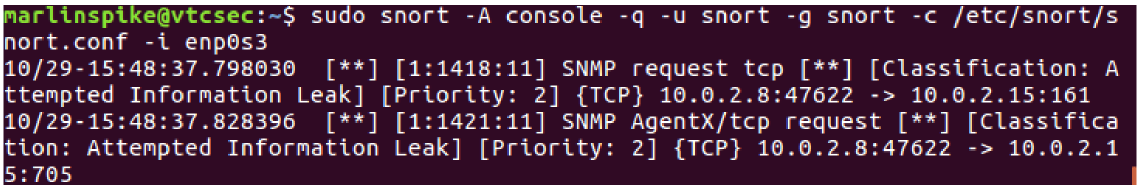
It can be noticed, for example, in the first line, that it’s related to TCP, it can come from any network (any source destination), the port number 21 coming to the home network, and a message with the description of the possible attack. We checked some other files to see their rules, what changes they produced, and what warning and warning massages they generated. After that, we used the command *sudo snort -T -c /etc/snort/snort.conf -i enp0s3*, this command checks if our Snort configuration is properly set:



```
marlinspike@vtcsec: ~
           Preprocessor Object: SF_IMAP  Version 1.0  <Build 1>
           Preprocessor Object: SF_DNS  Version 1.1  <Build 4>
           Preprocessor Object: SF_MODBUS  Version 1.1  <Build 1>
           Preprocessor Object: SF_REPUTATION  Version 1.1  <Build 1>
           Preprocessor Object: SF_GTP  Version 1.1  <Build 1>
           Preprocessor Object: SF_SDF  Version 1.1  <Build 1>
           Preprocessor Object: SF_SSH  Version 1.1  <Build 3>
           Preprocessor Object: SF_POP  Version 1.0  <Build 1>
           Preprocessor Object: SF_SMTP  Version 1.1  <Build 9>
           Preprocessor Object: SF_SIP  Version 1.1  <Build 1>
           Preprocessor Object: SF_FTPTELNET  Version 1.2  <Build 13>
           Preprocessor Object: SF_DCERPC2  Version 1.0  <Build 3>
           Preprocessor Object: SF_SSLPP  Version 1.1  <Build 4>

Snort successfully validated the configuration!
Snort exiting
```

Then we used the command *sudo snort -A console -q -u snort -g snort -c /etc/snort/snort.conf -I enp0s3* to start monitoring the attacks. After that, on Kali Linux, we used the command *nmap 10.0.2.15* to sniff the network of our Ubuntu machine. Then, as our Ubuntu machine was monitoring incoming attacks, it detected information leak due to the Kali Linux request:



```
marlinspike@vtcsec:~$ sudo snort -A console -q -u snort -g snort -c /etc/snort/s
nort.conf -i enp0s3
10/29-15:48:37.798030  [**] [1:1418:11] SNMP request tcp [**] [Classification: A
ttempted Information Leak] [Priority: 2] {TCP} 10.0.2.8:47622 -> 10.0.2.15:161
10/29-15:48:37.828396  [**] [1:1421:11] SNMP AgentX/tcp request [**] [Classifica
tion: Attempted Information Leak] [Priority: 2] {TCP} 10.0.2.8:47622 -> 10.0.2.1
5:705
```

It can be observed that there was a SNMP request and an information leak. We can also see the IP that requested the information *10.0.2.8* (Kali Linux) and the IP that was sniffed *10.0.2.15* (Ubunto).

After that, we also used (on Kali Linux against Ubunto) SPARTA, a GUI application that is used for network infrastructure penetration testing. And again, Ubunto's Snort was able to detect the attack. Those were the SNMP and ICMP based-attacks [1].

For the attack from a blacklisted source, we opened again the *etc/snort/snort.conf* file and uncommented the *preprocessor reputation* part (around line 506). We then used the three commands *sudo mkdir /etc/snort/rules/iplists*, *sudo touch /etc/snort/rules/iplists/black_list.rules*, and *sudo touch /etc/snort/rules/iplists/white_list.rules*, to create the blacklist and whitelist documents. In the black list document we added Kali Linux IP address. We again successfully tested our network by using *sudo snort -T -c /etc/snort/snort.conf -I eth0*. After that, we slightly changed the *etc/snort/snort.conf* file preprocessor reputation part, so that it started considering our blacklist. With those changes, the Ubunto machine prohibited Kali Linux sniffing, as Kali Linux was now blacklisted. That was the attack from a blacklisted source [2]

Part B

Part B consisted in creating a first-generation simple scanner, which searches for a string of bytes known to be from an identified virus. Usually those kinds of scanners have trouble finding polymorphic viruses, due to the polymorphic viruses' ability to change itself each time it replicates. The programming language JavaScript alongside with the Node.js runtime environment were used.

The program firstly read the provided file containing the viruses' definitions. It finds a variety of entries that contains possible viruses' signatures. After that, the scanner scans every file in a specific (hardcoded) folder and in all of its subdirectories, in order to find if there are any infected files, that is, if there are any file that matches with the viruses' signatures. If it finds an infected file, the program renders the virus inactive by changing the first 8 bytes of the signature to another sequence. Then, it moves the file to another (hardcoded) quarantine folder. For every file rendered and moved to quarantine, it keeps the user informed by showing him which files were infected, confirming that the virus was render inactive, and it showing that the virus was moved to the specific quarantine folder.

Considering underlying assumptions, two have to be mentioned: the path to the directory was hardcoded, the scanner acted specifically in that folder. If improvement was necessary, the user should be able to specify the folder to be scanned. Moreover, the quarantine folder was also hardcoded, however, that may be a common characteristic of a standard antivirus program.

To make the software useful for a production environment, it should be able to scan all folders in the system, or allow the user to decide which folder to scan. Moreover, the viruses’ signatures would have to be constantly updated, so that new viruses would also be identified. Therefore, the software would need to be capable of self-updating. Furthermore, a better UI would have to be designed, so that users could see the progress of the scanner’s scan, decide if the viruses should go to quarantine, decide if they wanted to delete them, etc. Besides that, the developed software is still a pretty simple scanner, it’s not suitable for any kind of security environment. Even though it roughly considers polymorphic viruses, there are still many other types of malwares that it wouldn’t be able to identify or protected against, such as metamorphic viruses, rootkits, keyloggers, logic bombs viruses, terminate and stay resident viruses, boot sector infector viruses, etc. We currently have forth generation antiviruses, which are much more complex and efficient. In conclusion, many changes would have to be done to the software in order to make it useful to a production environment.

Reference

[1] Liang Yang, L. (2019). *Network Intrusion Detection Systems (SNORT)*. [online] YouTube. Available at: https://www.youtube.com/watch?v=iBsGSsbDMyw [Accessed 31 Oct. 2019].

[2] Liang Yang, L. (2019). *Network Intrusion Detection Systems (SNORT)*. [online] YouTube. Available at: https://www.youtube.com/watch?v=iBsGSsbDMyw [Accessed 31 Oct. 2019].