Snort exercise

Offensive Technologies 2021
Matteo Franzil <matteo.franzil@studenti.unitn.it>
Claudio Facchinetti <claudio.facchinetti@studenti.unitn.it>

November 25, 2021

Contents

1 Solution

1.1 Topology

This photo shows the configuration of the nodes.

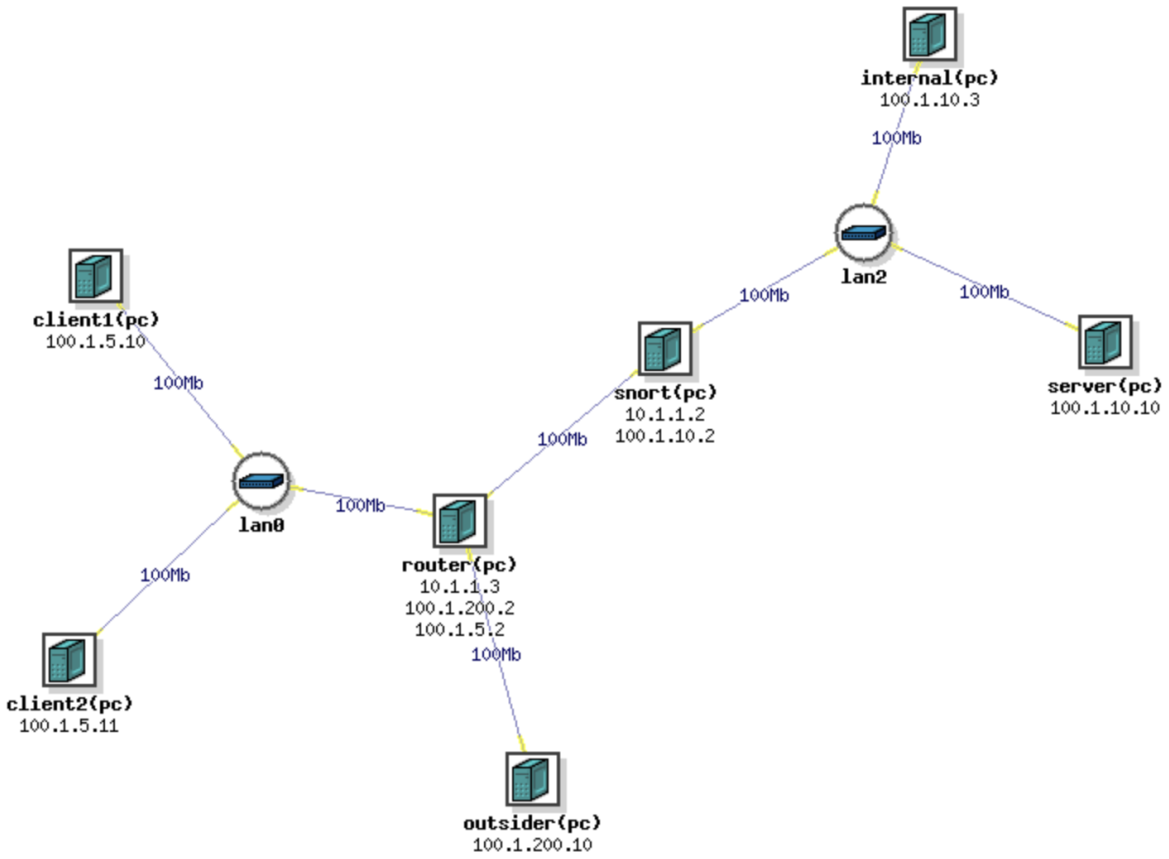

Figure 1 Network setup for the exercise.

1.2 Basic exercises

1.2.1 Start Snort Without Rules

Connect to `users.deterlab.net`. SSH into the `snort` experiment node.

Start Snort without any rules by entering the command `sudo snort --daq nfq -Q -v`.

Open another terminal and SSH again into `users.deterlab.net` and the `snort` node. In this terminal run `tcpdump` to capture the data going to and from the `client1`. This should be on the interface with an IP Address in the `10.1.1.0/24` range.

You can run tcpdump using the command:

```
# tcpdump -i [interface] -s 0 -w /tmp/dump.pcap
```

Code 1 Code for using tcpdump

Allow this to run for 30 seconds and then tcpdump by pressing `^C`. Let the Snort process continue running. Now run

```
# /share/education/SecuringLegacySystems_JHU/process.pl /tmp/dump.pcap
```

Code 2 Processing the obtained data.

This will produce a set of x, y coordinates where x is time and y is number of packets in that second.

Question 1.2.1. *Respond to the following points.*

1. *What happens to the traffic to client1 when Snort is not running?*
2. *Is this a good thing?*
3. *Based on Snort's output what can you say about the application? What port does it connect to?*
4. *Please attach a graph of the traffic over time to your answers*
5. *What does the "-Q" option do in Snort?*
6. *What does the "–daq nfq" option do in Snort?*

Answer. These are the answers.

1. When Snort is not running, all traffic gets dropped. We can see a slew of `SYN` packets being continuously sent by clients.
2. Not really. We can see that there are `client1`, `client2` and `outsider` that are continuously sending traffic to `server`, but it also gets dropped when Snort is not running. Since Snort is working in inline mode, all traffic is supposed to pass through the Snort bridge. When Snort is shut down, then the bridge is left hanging and traffic is automatically dropped.
3. The application accepts incoming connections on port 7777.
4. See the graph at the end of the numbered section.
5. The -Q option tells Snort to process packets in inline mode.
6. The –daq nfq option tells Snort to use the `nfq` DAQ module.

□

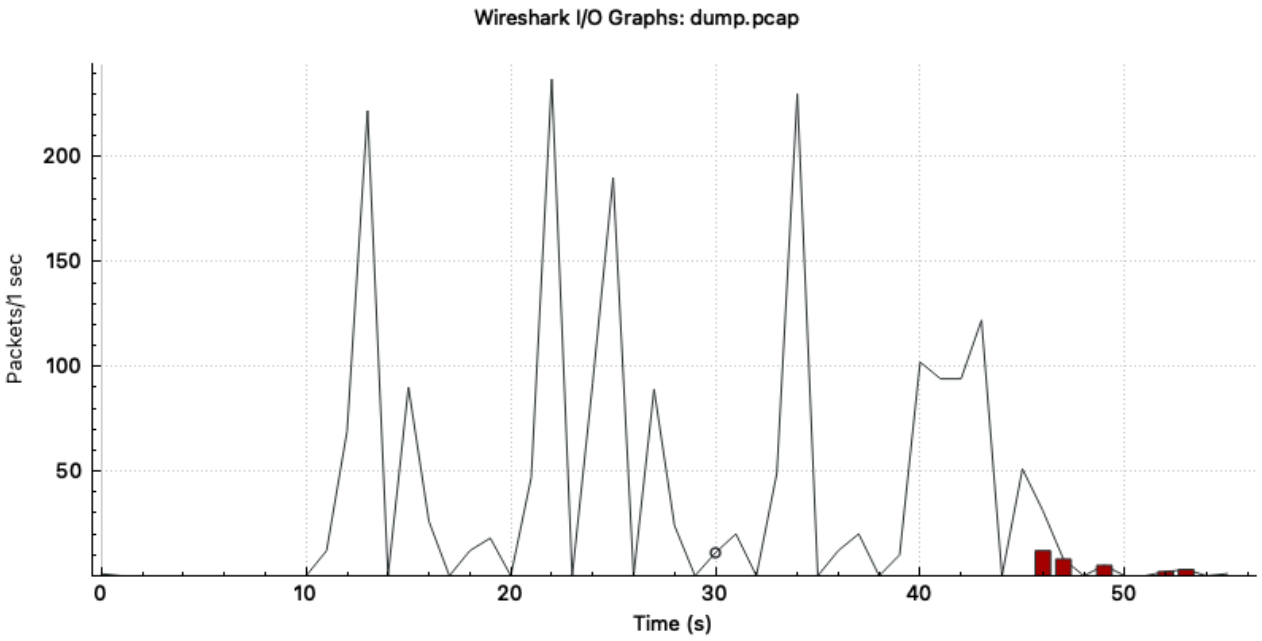

Figure 2 Traffic over time.

1.2.2 Analyze Network Traffic

Connect to the `router` node. Use `ifconfig` to determine which network interfaces connect to which network. Run `tcpdump` to capture the data going to the server. This should be on the interface with an IP Address in the `10.0.1.0/24` range.

You can run tcpdump with the same command as in Part 1. Let this run for around a minute before terminating it by pressing `^C`. Copy this data to your computer via SCP and open it using Wireshark.

Question 1.2.2. *Respond to the following points.*

1. *The request that the client sends the server is broken into four parts. What are these parts and what order does they appear in? How are these parts separated in the request?*
2. *Is this is a secure way for the client to send requests to the server? Explain your answer.*
3. *Can you recover one of the files sent by the server to a client? If so attach the file, a pcap the relevant packets and indicate which client this was sent to.*

Answer. These are the answers.

1. Figure 3 shows a snippet of the traffic. In our observations, each of the five different requests that the clients use a particular four-part string protocol in which the data is divided with the `!::!` separator. In particular, the first field contains a username, the second contains a password, the third contains the name of the file, the fourth is the actual content of the file preceded by a certain amount of *junk*.
2. It is not. Disregarding the fact that the connection is not encrypted - so all the payload is in clear and can be easily read with some simple traffic sniffing - the third field contains the name of the file that is attached to the message. This easily allows network sniffers such as Snort to intercept and block such requests if they contain sensitive data, without resorting to file parsing. However, this also allows man-in-the-middle attackers to obtain information about what the client is querying before the data is even retrieved.
3. See the attached `pcap` file and Figure 4. After selecting TCP stream 56, I used the *Follow TCP Stream* Wireshark tool to visualize the data exchanged between the client and the server.

□

No.	Time	Source	Destination	Protocol	Length	Info
431	11.3083…	100.1.5.11	100.1.10.10	TCP	74	59704 → 7777 [SYN] Seq=0 Win=64240
432	11.3094…	100.1.10.10	100.1.5.11	TCP	74	7777 → 59704 [SYN, ACK] Seq=0 Ack=1
433	11.3098…	100.1.5.11	100.1.10.10	TCP	66	59704 → 7777 [ACK] Seq=1 Ack=1 Win=
435	11.3188…	100.1.5.11	100.1.10.10	TCP	70	59704 → 7777 [PSH, ACK] Seq=1 Ack=1
438	11.3199…	100.1.10.10	100.1.5.11	TCP	66	7777 → 59704 [ACK] Seq=1 Ack=5 Win=
453	11.3625…	100.1.5.11	100.1.10.10	TCP	493	59704 → 7777 [PSH, ACK] Seq=5 Ack=1
454	11.3638…	100.1.10.10	100.1.5.11	TCP	66	7777 → 59704 [ACK] Seq=1 Ack=432 Wi
468	11.4745…	100.1.10.10	100.1.5.11	TCP	70	7777 → 59704 [PSH, ACK] Seq=1 Ack=4
469	11.4750…	100.1.5.11	100.1.10.10	TCP	66	59704 → 7777 [ACK] Seq=432 Ack=5 Wi
532	12.4509…	100.1.10.10	100.1.5.11	TCP	451	7777 → 59704 [PSH, ACK] Seq=5 Ack=4
535	12.4514…	100.1.5.11	100.1.10.10	TCP	66	59704 → 7777 [ACK] Seq=432 Ack=390
539	12.4522…	100.1.5.11	100.1.10.10	TCP	66	59704 → 7777 [FIN, ACK] Seq=432 Ack
540	12.4522…	100.1.10.10	100.1.5.11	TCP	66	7777 → 59704 [FIN, ACK] Seq=390 Ack
541	12.4526…	100.1.5.11	100.1.10.10	TCP	66	59704 → 7777 [ACK] Seq=433 Ack=391
543	12.4542…	100.1.10.10	100.1.5.11	TCP	66	7777 → 59704 [ACK] Seq=391 Ack=433

Figure 3 Snippet of a TCP stream from the first point.

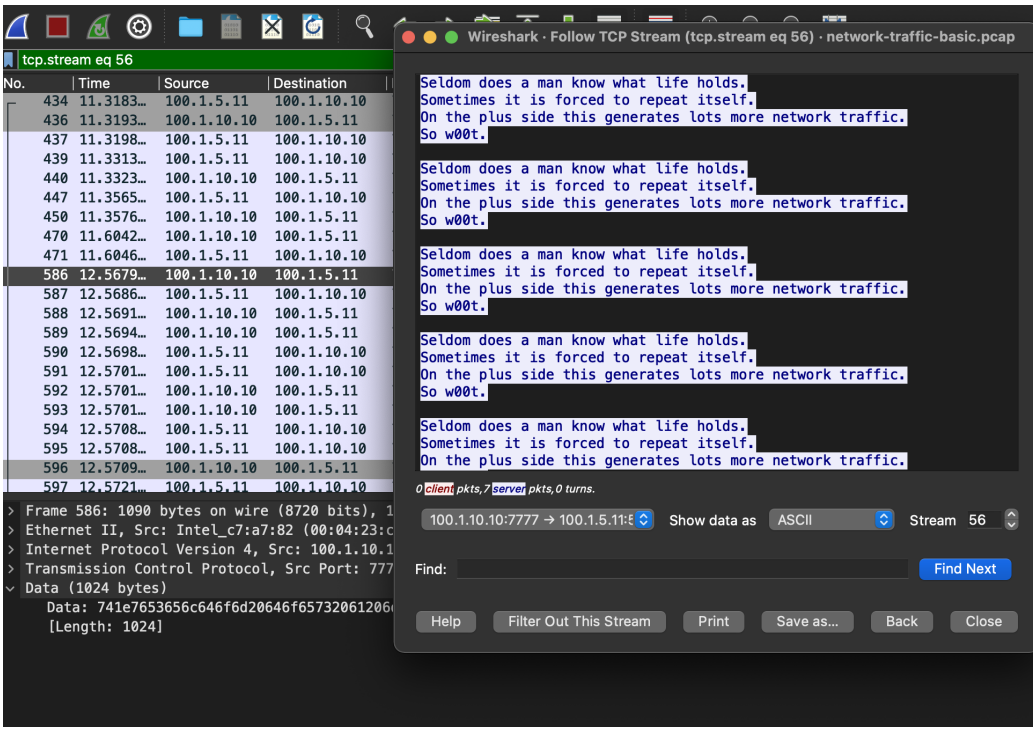

Figure 4 Following a TCP stream.

1.2.3 Write Rules to Guard Against Simple Requests

In that terminal where Snort is running stop it with `^C`. Write a configuration file for Snort using the command, named `snort.config`. Add the following Snort rule that prevents .xml files from being sent out.

```
reject tcp 100.1.0.0/16 ANY -> 100.1.10.10 [Port from Question 3]
    (msg: "XML Read Attempt Detected"; sid:1; content:".xml";)
```

Code 3 Code to add to the snort.config

Using this rule as an example write a rule that prevents classified data from being sent to the outsider computer, but does not prevent it from being sent to any other computers. Now you must make a directory for Snort alerts, called `alerts`. Start Snort using your new rule with the command:

```
# snort --daq nfq -Q -c snort.config -l alerts
```

Code 4 Code to start snort.

Log onto `client1`, `client2` and `outsider` to see if these rules worked. If you aren't sure go to the folder `/home/test` and delete the existing .txt and .xml files then see which ones are recreated. You can also run the program FileClient manually to find out. Questions:

Question 1.2.3. *Respond to the following points.*

1. *What rule did you use to secure the "classified" file?*
2. *Capture and compare the network traffic for the server when filtering these results using your configuration file and when no file is used. Attach the graph showing packet rate over time for both of these cases to your submission.*
3. *Can you think of any others files or extensions that should be filtered against?*

Answer. These are the answers.

1. `reject tcp 100.1.200.10 ANY -> 100.1.10.10 7777`
   `(msg: "Data Exfiltration"; sid:1; content:"classified";)`
2. See Figure 5 and 6 for the packets graphs. The difference isn't noticeable: after all, we blocked $\frac{1}{5}$ of the traffic coming from one host out of three, so $\frac{1}{15}$ of the total traffic. The graph isn't enough descriptive to appreciate the difference.
3. By looking at the downloaded documents, `users.txt` looks like a good candidate to be filtered out.

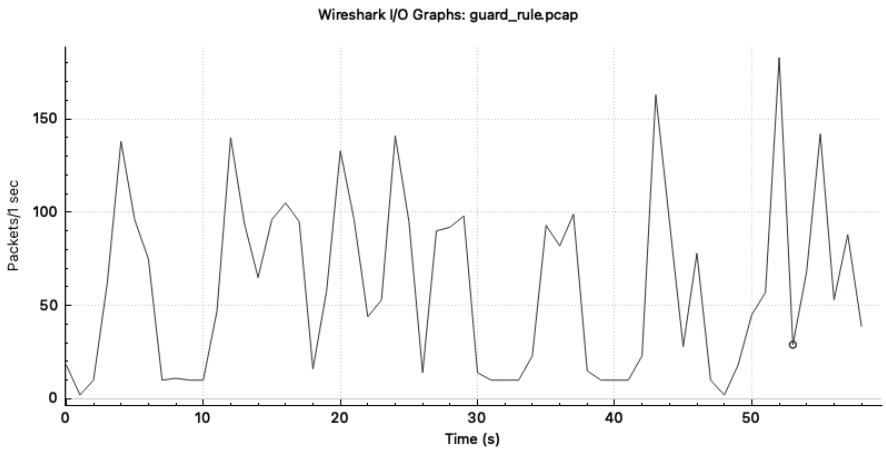

Figure 5 Graph of TCP packets with the enabled rule.

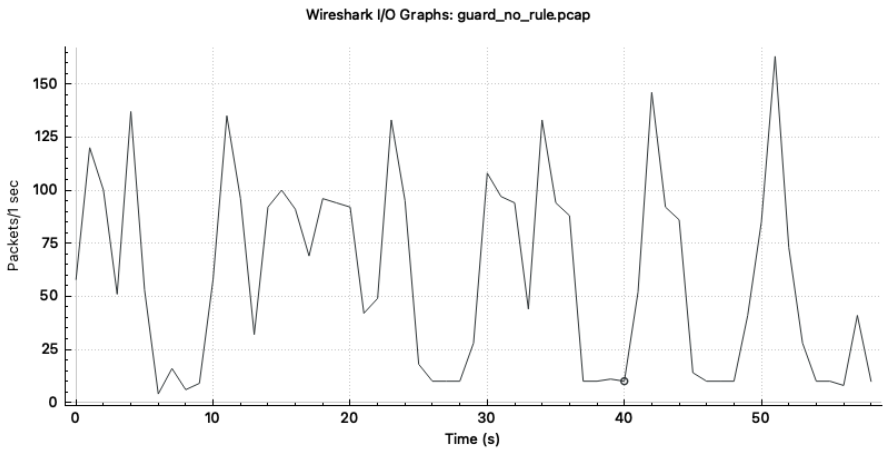

Figure 6 Graph of TCP packets without the enabled rule.

□

1.3 Intermediate tasks

1.3.1 DOS defense

Connect to the `server` node and use the `ip addr` command to find out which is the network interface with the IP address `100.1.10.10`. Now connect to the `router` node and use `ip addr` to determine the network interface which has an IP address in the range `10.0.1.0/24`.

Run `tcpdump` on both the `server` and `router` nodes on the discovered interfaces to collect traffic when no attack is going on.

Now connect to the `client1` node and run the following script for 100 seconds to simulate the attack.

```
#!/bin/sh
/share/education/TCPSYNFlood_USC_ISI/install-flooder
sudo flooder --src 100.1.5.10 --srcmask 255.255.255.255 --dportmin 7777
    --dportmax 7777 --highrate 100000 --hightime 100
    --dst 100.1.10.10 --dstmask 255.255.255.255
```

Code 5 Command to start the attack

As the attack is ongoing, use again `tcpdump` to capture traffic on both the `router` and `server` machines. Now append the following rules to the `snort.config` file so that the UDP flooding attack can be avoided.

```
rate_filter gen_id 1, sig_id 10, track by_rule, count 100,
    seconds 1, new_action drop, timeout 5
event_filter gen_id 1, sig_id 10, track by_rule, count 20,
    seconds 1, type threshold
pass udp any any -> 100.1.10.10/32 7777 (msg: "Possible flooding"; sid: 10;)
```

Code 6 Rule for UDP Flooding

With this rule you are instructing Snort to allow any traffic to the server machine on port 7777; when 100 packets per second are matched then the rate filter starts dropping them. This behaviour remains valid until the threshold imposed by the event filter is higher than the actual number of packets.

Repeat the capturing procedure described above two more times: one with no attack going and one while the attack is active.

Question 1.3.1. *Respond to the following points.*

1. *Collect the traffic at the server and the router when there is and is not an attack with rules in place that only guard against the attacks mentioned in the basic exercises. Create a graph of this traffic over time.*

2. *Repeat the previous step but now with the rate filtering rules enabled.*

3. *For a DOS attack should rate filtering rules be paired with event filtering rules?*

4. *Try changing the new action in the rate filter to "reject" instead of "drop". What does this do to the traffic and why do you think this is? Is this a good or a bad thing?*

5. *Check which interface Snort connects to the router to using* `ip addr`*. Once you have this information try the above test against while specifying that interface instead of using the default value. This argument will look something like –daq-var device=eth1. Did this change your results. If so attach a graph and explain the change occurred.*

6. *Are there any changes you can make outside of Snort to help guard against this attack? If so what are they?*

7. *Try running the FileClient program from client2 while this attack is underway. What happens when you have the various rule sets configured?*

Answer. These are the answers.

1. Graphs can be seen in Section 1.3.2.

2. Same as before.

3. Yes, they should always be paired with event filtering rules, otherwise there would be no way of making the new action imposed by the rate limiter revert to the normal behaviour.

4. When changing the new action to reject instead of drop a RST packet is sent back to the sending machine. If the attacker is not spoofing then nothing bad should happen; if the attacker is spoofing then we might flood the spoofed IP address with a lot of RST packets. In any case changing the behaviour to reject creates more traffic on the network and possibly discloses information, therefore it would be better to stay with drop.

5. When specified the `device` variable specifies which interface is used by Snort to capture network traffic. In this case even if we specify the interface connected to the router we would observe little changes: if we make the `internal` machine use `snort` as gateway then the only traffic that is no longer captured is the one which from/to the `internal` node.

6. Running the FileClient program from the `client2` node while the attack is in progress results in a higher waiting time before data is received, but it is still able to receive data. The delay is probably caused by the fact that the links are flooded with a high number of packets per second.

□

1.3.2 DOS defense graphs

This section shows the graphs requested in the DOS defense exercise (Section 1.3.1) .

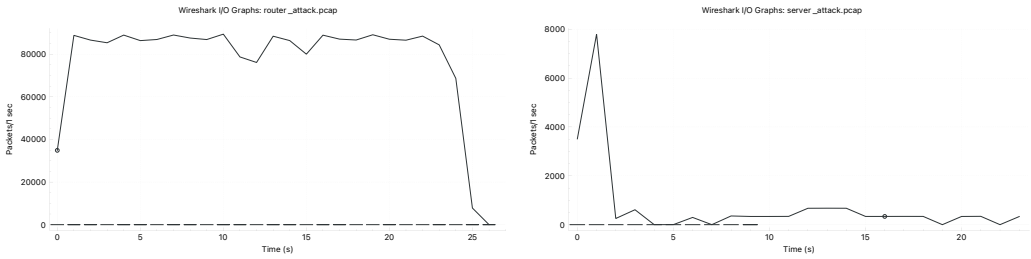

Figure 7 Graph at router and server machines, with **an attack** ongoing and **no** rules.

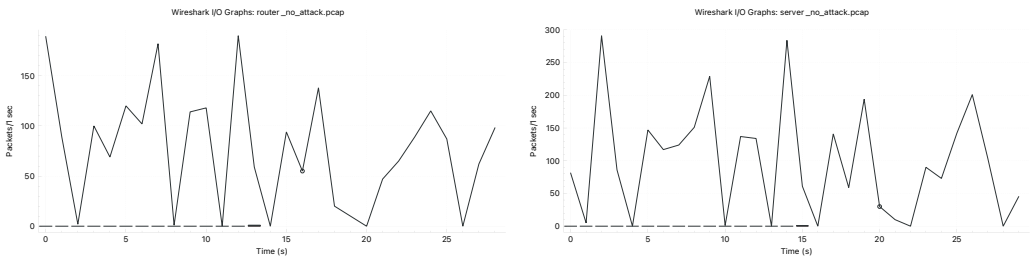

Figure 8 Graph at router and server machines, with **no** attacks ongoing and **no** rules.

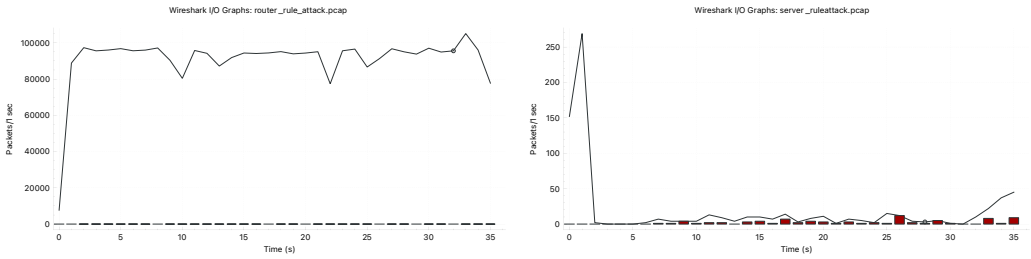

Figure 9 Graph at router and server machines, with **an attack** ongoing and **active** rules.

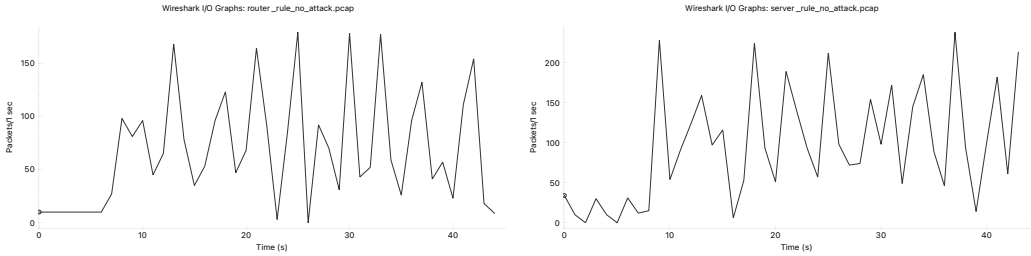

Figure 10 Graph at router and server machines, with **no** attacks ongoing and **active** rules.

1.3.3 Secure Traffic on the Network

Connect to the `internal` machine and run the following command to make the traffic from this machine to the `server` machine flow through `snort`.

```
# route add -host server gw snort
```

Code 7 Code for replacing the direct route to the server.

Verify that the changes had effect by running `traceroute server` onto the `internal` machine.

```
traceroute to server (100.1.10.10), 30 hops max, 60 byte packets
 1  * * *
 2  server-lan2 (100.1.10.10)  0.474 ms  0.477 ms *
```

Code 8 Result of traceroute on internal machine

Question 1.3.2. *Respond to the following points.*

1. *Based on the traffic you analyzed what changes could be made to the network to enhance the security of communications coming from client1, client2 and outsider?*
2. *What software packages would this require and where should these be installed? Would this cause problems for Snort?*
3. *How should the server be configured to prevent internal attacks?*
4. *Would this require you to change your Snort configuration in any way?*

Answer. These are the answers.

1. Right now the communication is completely in plain text and anyone could easily steal usernames and passwords to grab the files. A possible solution to this would be using encryption to secure communications going from client to server and vice versa.

2. The implementation of cryptography would require some changes to the protocol, meaning that both the client and the server need to be rewritten to support encryption and decryption. This would however be a pain in the neck for Snort: it can no longer check for package content since everything is encrypted and therefore Snort will simply act as a stateful firewall instead of a proper NIDS/NIPS.

3. In order to be safe against internal attacks the server should require some kind of strong authentication, like physical tokens, in order to perform administrative operations and all the traffic should be filtered through Snort so that there is no traffic left unchecked.

4. This would probably require some new rules on Snort (blocking potential attacks, allowing protocols, etc) and probably another instance of Snort: if we want to be completely sure an idea could be placing a Snort instance which filters traffic coming from the outside world directed to the internal network and another one which filters intranet traffic. The main idea is that the two instances can have different configurations to prevent different threats. An alternative could be a single instance which filters all the traffic from and to the service, but this would become a single point of failure: if it crashes for any reason the server would be no longer accessible.

□

1.4 Advanced Tasks

1.4.1 Code Execution Vulnerability

As a first thing download from the `server` machine the file `FileServer.jar` using the `scp` util.
Now install any Java decompiler, such as `JD-GUI` and open the downloaded file.
Open the `jar` file using the Java decompiler and look for a line of code which looks like the one reported below.

```
Runtime.getRuntime.exec(dividedData[2]);
```

Code 9 Line causing the code execution

That is the line where it is possible to perform command execution; to do so use the following script one of the client machines. It is fundamental that the payload is at least 2001 characters long.

```
#!/bin/bash

java -jar FileClient.jar zaqrbo[...]ooob
    zaqrpassword server sudo%32shutdown%32now
```

Code 10 Code to shutdown the server remotely

Question 1.4.1. *Respond to the following points.*

1. *What are the conditions required for this attack to take place?*
2. *Create a Snort rule to defend against this attack. You may want to be use pcre instead of content for this rule.*
3. *What effect does this rule have on legitimate traffic?*

Answer. These are the answers.

1. In order to take place the combination of username, password, file name, random junk inserted by the client and separators must be at least 2001 characters long; this string must also contain at least a substring which matches the regular expression `z.0,2a.0,2q.0,2e`.
2. In order to defend against this attack append the following rule to the `snort.config` file.

   ```
   drop tcp any any -> 100.1.10.10/32 7777
       (msg:"RCE Attempted"; pcre:"/z.{0,2}a.{0,2}q.{0,2}r/"; sid:1000000;)
   ```

   Code 11 Snort rule to defend against RCE

3. This rule will block RCE but will also block any legitimate request which will any any username, password, file name or random stuff appended at the end matching that particular regular expression.

□

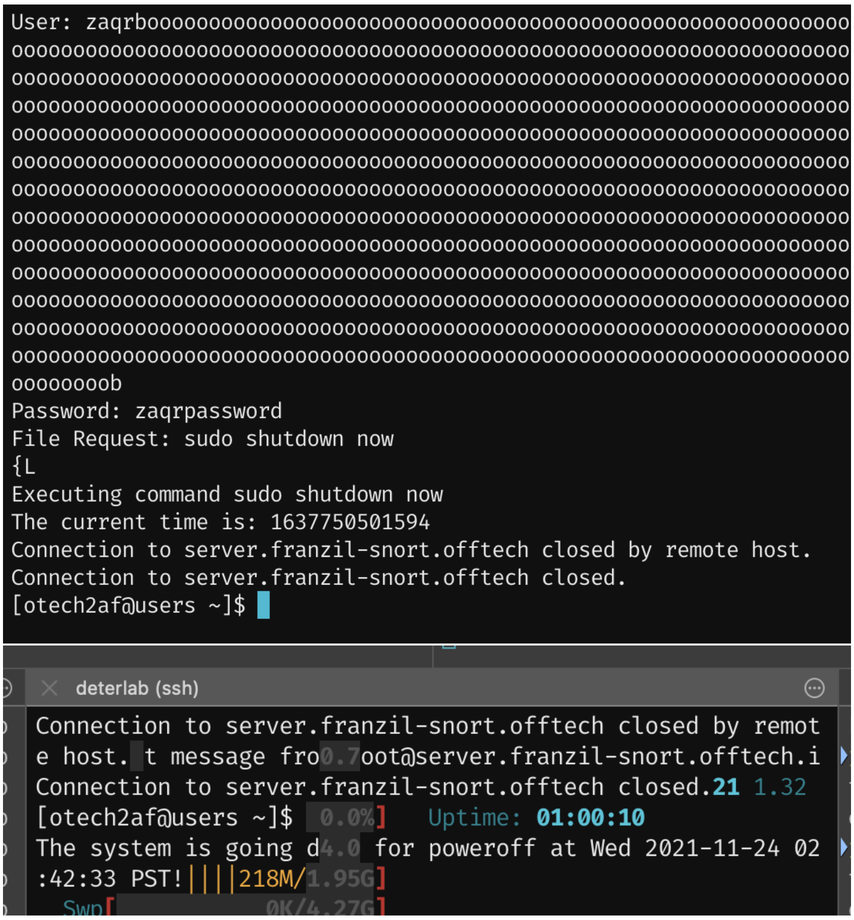

Figure 11 Shutting down the server with the exploit.

1.4.2 Defend Against ASCII Encoding

An additional "feature" was recently brought to the attention of the security team: the application has a limited ability to before ASCII decoding. This decoding is triggered whenever the server encounters a % followed by a number between 0 and 255 at which point the sequence is converted to a single ASCII character.

Now go to the following directory:

`/usr/local/snort-2.9.2.2/src/dynamic-examples/dynamic-preprocessor`

This contains an example Snort preprocessor. Snort preprocessors allow for advanced data manipulation to normalize data and perform other functions that normal rules cannot. All Snort preprocessors are coded in C. Copy the include folder using the following command:

```
# cp -rf /usr/local/snort-2.9.2.2/src/dynamic-preprocessors/include/ ..
```

Code 12 Code for including the folder.

You should now be able to compile and install this preprocessor using the command:

```
# make && make install
```

Code 13 Make command.

Once this has been installed you can use this preprocessor in Snort. To do this edit your configuration file by adding the following lines at the top of the file which tell Snort to auto-generate the decoder rules so you don't have to, the location of the preprocessor library and the preprocessor you wish to use along with all of its arguments (in this case just the port to use):

```
config autogenerate_preprocessor_decoder_rules
dynamicpreprocessor directory /usr/local/lib/snort_dynamicpreprocessor/
preprocessor dynamic_example: port 7777
```

Code 14 Code for adding the preprocessor to the configuration file.

Once you have made these changes try to run Snort using your configuration file. At this point it should behave exactly as before, but it now has a preprocessor loaded.

Question 1.4.2. *Respond to the following points.*

1. *Were you able to bypass your existing rules because of this feature? If so what input strings did you use?*
2. *Can you think of a content rule to effectively defend against an attack that uses this feature? Would this affect legitimate traffic?*
3. *Snort includes support for user written preprocessors that can render data for Snort's other rules. How would the use of a preprocessor help with this task?*
4. *Write a preprocessor to help with this task. Please attach all of the functions you used and the snort.config file that called the preprocessor.*

Answer. These are the answers.

1. Since `zaqr`, an offending substring in the preivous output, can be encoded as `%122%097%113%114`, then one could do a request such as:

   ```
   java -jar FileClient.jar \
       %122%097%113%114bo[...]b %122%097%113%114password \
       server sudo%32shutdown%32now
   ```

   which would pass through the filters and execute the vulnerability.

2. One idea would be to create a content rule (pcre), with a % followed by any number, but it would apparently risk being in the way of actual strings containing legitimate % symbols followed by numbers (e.g. clients visiting files with actual % characters in the filenames). However, since the server would do the parsing anyway

3. The preprocessor would be able to parse the payloads of incoming packets, and systematically substitute any % followed by a number $[0, 255]$ with the corresponding ASCII code. At this point, any malevolent request would be easily spotted and neutralized by the previous rule.

4. See the attached files. The edited function was the `ExampleProcess` one, which processes packet data before other rules have a chance to access it.

□