



# UNIVERSITÀ DI TRENTO

Network Security Report

## LAB 9 NIDS - SNORT

Alessandro Brighenti - 223809

Matteo Liberato - 220233

Francesco Pavanello - 220372

Academic Year 2020/2021

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Snort</b>	<b>4</b>
2.1	IDS . . . . .	4
2.2	IPS . . . . .	4
<b>3</b>	<b>Snort Rules</b>	<b>5</b>
3.1	Rule Header . . . . .	5
3.1.1	Action . . . . .	5
3.1.2	Protocol . . . . .	5
3.1.3	IP addresses and ports . . . . .	5
3.1.4	Direction Operator . . . . .	5
3.2	Rule Options . . . . .	6
3.2.1	General Options . . . . .	6
3.2.2	Payload Detection Options . . . . .	6
3.2.3	Non-Payload Detection Options . . . . .	8
3.2.4	Post-Detection Options . . . . .	9
3.3	Event Processing . . . . .	9
3.3.1	rate_filter . . . . .	9
3.3.2	event_filter . . . . .	10
<b>4</b>	<b>Environment Setup</b>	<b>11</b>
4.1	Network Configuration . . . . .	11
4.2	Snort Installation . . . . .	15
4.3	Snort Configuration . . . . .	16
<b>5</b>	<b>Exercises</b>	<b>18</b>
5.1	PING . . . . .	19
5.1.1	Exercise . . . . .	19
5.1.2	Solution . . . . .	19
5.2	Port Scanning . . . . .	20
5.2.1	Exercise . . . . .	20
5.2.2	Solution . . . . .	20
5.3	Buffer Overflow . . . . .	21
5.3.1	Exercise . . . . .	21
5.3.2	Solution . . . . .	21
5.4	SQL Injection . . . . .	22
5.4.1	Exercise . . . . .	22
5.4.2	Solution . . . . .	22
5.5	Private File Request . . . . .	24
5.5.1	Exercise . . . . .	24
5.5.2	Solution . . . . .	24
5.6	SYN Flood Attack . . . . .	25
5.6.1	Exercise . . . . .	25
5.6.2	Solution . . . . .	25

<b>Code</b>	<b>26</b>
A    Server . . . . .	26
A.1    server.py . . . . .	26
A.2    sourceC.cc . . . . .	27
B    Attacker . . . . .	28
B.1    port.scan.sh . . . . .	28
B.2    flooder.sh . . . . .	28
<b>References</b>	<b>29</b>

# 1 Introduction

This laboratory has the scope of showing how Snort works both as NIDS and IPS.

After the necessary introductory notions about Snort and how its rules work, we provide some exercises in order to put in practice what we learnt during the theoretical lessons. These exercises consist in the detection or mitigation of some possible attacks.

The environment in which the laboratory is conducted is composed of three virtual machines: a server where a flask application is running, a router with Snort installed in it and an attacker machine where there are scripts that automate some attacks. The server and the attacker are in two different networks among which the router acts as a gateway. In most cases, for the scope of this laboratory, we will consider the server as a legacy system, where the only mitigation of the attacker can be performed only on an external machine (the router), otherwise other more effective solutions could be applied directly on the server.

The key takeaway from these exercises is that monitoring and protecting a network is a daunting task, but if done correctly can also make the job of attackers very difficult.

## 2 Snort

Snort is a free open source software, first created in 1998 by Martin Roesch and then bought by Cisco in 2013, used as Intrusion Detection System (IDS) and Intrusion Prevention System (IPS).

This program has the ability to analyze in real time the traffic and to perform packet logging in IP networks, to do this it checks the packet against all of its rules and, if the rule matches, actions are taken, ranging from issuing alerts to dropping the packet, depending on the mode and setup of the rules. Snort as an instrument can be used in 3 different modes:

- **Packet Sniffer**, which reads packets that are moving through the network and shows them.
- **Packet logger**, which logs the packets to disk.
- **Network Intrusion Detection System (NIDS)**, which uses the signature to perform analysis of network traffic, this last one is the one that we focused on the most.

### 2.1 IDS

An IDS is a software that monitors networks or systems for malicious activities and produces reports, if the software is placed in a specific point in the network to monitor traffic then it's called NIDS (Network Intrusion Detection System).

To start Snort as an IDS it's necessary to use this command:

```
sudo snort -c /etc/snort/snort.conf -A console
```

### 2.2 IPS

When used as IPS Snort gains the capability of undertaking an active role, sending a warning and dropping said packets in case they match the rule, stopping them from reaching their destination.

To start Snort as an IPS it's necessary to use this command:

```
sudo snort -c /etc/snort/snort.conf -A console -Q
```

This command is identical to the one used to start Snort with the intention of using it as an IDS, with an addition, the flag `-Q` at the end of the string. This flag's purpose is to start Snort in *inline mode*, this means that the packets will pass through Snort, which will act as a bridge between the source and the destination of the message, allowing it to drop the packet if the rule is matched.

Thanks to these abilities, this program can be used to detect various kind of probes and attacks.

## 3 Snort Rules

In this chapter we reported the notions we thought are the most important in order to move the first steps in discovering the power of Snort. We took inspiration from the Snort Manual [3].

Snort could perform packet inspection through rules. They are written in a simple, flexible language. It provides itself some rules written by the community of users, as well as rules tested and approved by Cisco. Rules could also be written from scratch. These "custom" rules would be written in file located at: `/etc/snort/rules/local.rules`

Rules format is:

`[action][protocol][sourceIP][sourcePort]→[destIP][destPort] ([Rule Options])`

The part in blue is the rule header and the part at the end contains the rule options.

### 3.1 Rule Header

The rule header contains information about packet sender and packet recipient, as well what to do when a packet with indicated attributes shows up.

#### 3.1.1 Action

*action* field specifies the action to take when the rule matches a packet:

- **action** generates an alert using the selected alert method (usually the console) and then logs the packet
- **log** logs the packet
- **pass** ignores the packet
- **drop** blocks and logs the packet
- **reject** blocks the packet, logs it and then send a TCP reset (TCP) or an ICMP port unreachable (UDP)
- **sdrop** blocks the packet without logging it

#### 3.1.2 Protocol

*protocol* specifies what type of traffic the rule will analyze. Possible values are TCP, UDP, IP, ICMP.

#### 3.1.3 IP addresses and ports

IP addresses of packet source and destination could be specified as a fixed IP (e.g. 192.168.1.1) or a CIDR block (e.g. 192.168.1.0/24). In the latter case, the rule will try to match all packets that go to hosts of the specified network.

Port numbers are specified as numbers. They could be also a range of ports: `1:1024` will match packets coming to/arriving from ports between 1 and 1024.

IP addresses and port numbers could be substituted by the keyword **any** to match any possible IP or port.

#### 3.1.4 Direction Operator

The direction operator `→` indicates the direction of traffic. IP and port on the left side of the arrow is source, IP and port on the right side of the arrow is destination.

There is also a bidirectional operator, `<>`, that tells Snort to consider address port pairs as source or destination, depending of traffic orientation

## 3.2 Rule Options

Rule options are the heart of Snort's intrusion detection engine. All Snort rule options are separated from each other using a semicolon (;).

We will use some rules options from all the 4 categories: **general**, **payload detection**, **non-payload detection** and **post-detection**.

### 3.2.1 General Options

These options provide informations about the rule and they are not related to content of packets.

Some useful general options are:

- **msg:** "<message text>"; is the text to visualize when a rule matches a packet
- **sid:** <Rule id>; specifies the unique ID of the rule. A sid > 1000000 is necessary for local rules, to avoid conflicts with community rules
- **rev:** <Revision integer>; is the number of revision of the rule

### 3.2.2 Payload Detection Options

Payload options allow Snort to look for data inside the packet payload and can be inter-related.

#### content

Content option is one of the most important features of Snort. It allows the user to search for specific content in packet payload. The search is normally performed case **sensitive**.

The value of content option could be:

- A string of text, to be searched literally
- Binary data, represented as bytecodes (hexadecimal numbers), enclosed in pipes and separated by a whitespace (e.g. |code1 code2|)
- A mix of text and bytecodes

If the value of option is prefixed with a !, the rule matches payloads without the specified string:

- **content:** "&="; matches a packet if its payload contains the "&=" string
- **content:** "|26 3d|"; also matches "&=", because 26 is the ascii hex number for & and 3d is the ascii hex number for =
- **content:** "!&="; matches a packet if its payload *doesn't* contain the "&=" string

#### Content Option Modifiers

There exist some keywords (to be specified immediately after content rule) to modify the behaviour of content:

- **nocase;** is used to make content search in **case insensitive** mode
- **depth <X>;** content needs to be searched in only the first *X* bytes of the payload
- **offset <X>;** content needs to be searched after the first *X* bytes of the payload

It is important to notice that more than one *content* option could be written in a rule.

Suppose that a rule contains two *content* options. The rule matches a packet if it contains in its payload:

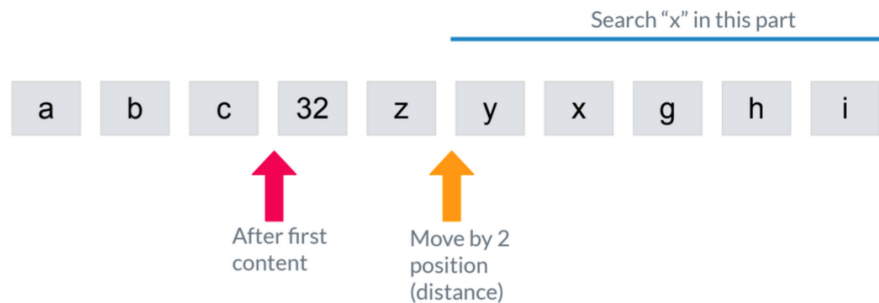
- the value of first *content* option
- the value of second *content* option, starting from the end of the first match

There are also some modifiers related to this:

- **distance <X>;** value of second *content* option needs to be searched *X* bytes after the end of first match

- **within** <X>; value of second *content* option needs to be searched in at most *X* bytes after the end of first match. If *within* is used in conjunction with *distance*, value of second *content* option needs to be searched in at most *X* bytes after the position given by *distance*

The following is an example of usage of more than one *content* option:



String to match and pointer (32 is ascii code for the space character)

Suppose we want to match a packet in which there is the string "abc zyxghi" in the payload. A rule could contain these options:

```
content: "abc"; content:"x"; distance: 2; within:5;
```

In the figure above, we can see how the evaluation works. We can imagine to have a pointer that moves along the string in the payload.

Firstly, the rule matches "abc", thus the pointer moves after the letter "c". Now, there is another content option, that will try to search a "x" in the remaining of the payload. But there are also two modifiers: *distance* and *within*. Because of *distance: 2*, the pointer moves 2 bytes (characters) ahead. Because of *within: 5*, it will start searching for a "x" in the following 5 bytes (i.e. characters)

### pcre

This keyword allows us to write rules that match packets payload using perl compatible regular expression, which is a library that implements syntax and semantics of regular expression as in Perl 5.

With regular expression (regex), patterns are used to determine if content in payload has (or doesn't have) some characteristics.

In regex, some characters have special meaning and they are called metacharacters. They are:

{ } [ ] ( ) ^ \$ . | \* + ? \

To match these characters literally, they need to be escaped with \.

\n and \t could be used to specify a newline character and a tabulation character.

. matches any character except newline.

The "|" is used as an OR between two words: (this|that) matches a string if it contains "this" or "that".

\xnn can be used to specify a character that has *nn* as hexadecimal ascii code.

Full list of metacharacters could be find here

[] are used to define character classes:

- [0-9] is the class of digits
- [abc] is the class composed of *a*, *b*, *c*
- \w is the class of any alphanumeric character plus underscore

The full list of predefined character classes can be found [here](#)

In regex there are also quantifiers, which are used to specify that a portion of the pattern needs to match the content of the payload a certain number of times:

- \* matches 0 or more times



- `+` matches 1 or more times
- `?` matches 0 or 1 time
- `{n}` matches exactly n times
- `{n,}` matches at least n times
- `{,n}` matches at most n times
- `{n,m}` matches at least n but not more than m times

It is important to notice that, when a quantifier is used after a character class, it says how many occurrences of the character class need to be matched.

In snort, `pcre` option format is:

```
pcre: [!]" /regex/modifiers".
```

Modifiers are used to modify the behaviour of the regex. The most useful modifiers are:

- `i` to perform a case insensitive matching
- `s` to include newlines in dot metacharacter
- `x` to ignore whitespace data characters in pattern except when escaped or inside a character class (to have more readable regex)

Usually, `pcre` option is used in conjunction with `content`. The latter works as a fast-pattern matcher to filter out non-matching packets.

### 3.2.3 Non-Payload Detection Options

These options look for non-payload data.

#### **itype**

The `itype` keyword is used to check for a specific ICMP type value. The format is:

```
itype:min<>max;
itype:[<|>]<number>;
```

#### **flags**

The `flags` keyword is used to check if specific TCP flag bits are present.

The following bits may be checked:

- **F** - FIN - Finish (LSB in TCP Flags byte)
- **S** - SYN - Synchronize sequence numbers
- **R** - RST - Reset
- **P** - PSH - Push
- **A** - ACK - Acknowledgment
- **U** - URG - Urgent
- **C** - CWR - Congestion Window Reduced (MSB in TCP Flags byte)
- **E** - ECE - ECN-Echo (If SYN, then ECN capable. Else, CE flag in IP header is set)
- **0** - No TCP Flags Set

The following modifiers can be set to change the match criteria:

- `+` - match on the specified bits, plus any others
- `*` - match if any of the specified bits are set
- `!` - match if the specified bits are not set

The format is:

```
flags:[!|*|+]<FSRPAUCEO>[,<FSRPAUCE>];
```

### 3.2.4 Post-Detection Options

These options are rule specific triggers that happen after a rule has “fired.”

#### **detection\_filter**

`detection_filter` defines a rate which must be exceeded by a source or destination host before a rule can generate an event. It has the following format:

```
detection_filter: track <by_src|by_dst>, count <c>, seconds <s>;
```

- **track** defines if the rate is tracked either by source IP address or destination IP address. This means count is maintained for each unique source IP address or each unique destination IP address.
- **count** specifies the maximum number of rule matches in s seconds allowed before the detection filter limit to be exceeded. It must be nonzero.
- **seconds** sets the time period over which count is accrued. The value must be nonzero.

Snort evaluates a `detection_filter` as the last step of the detection phase, after evaluating all other rule options (regardless of the position of the filter within the rule source). At most one `detection_filter` is permitted per rule.

## 3.3 Event Processing

Besides the `detection_filter`, there are other mechanisms to tune event processing to suit your needs. We reported here only the ones that will be useful during the exercises.

### 3.3.1 rate\_filter

`rate_filter` provides rate based attack prevention by allowing users to configure a new action to take for a specified time when a given rate is exceeded. Multiple rate filters can be defined on the same rule, in which case they are evaluated in the order they appear in the configuration file, and the first applicable action is taken. It is used as standalone configurations (outside of a rule) and has the following format:

```
rate_filter gen_id <gid>, sig_id <sid>, track <by_src|by_dst|by_rule>, \  
count <c>, seconds <s>, new_action alert|drop|pass|log|sdrop|reject, \  
timeout <seconds> [, apply_to <ip-list>]
```

- **gen\_id** specifies the generator ID of an associated rule. `gen_id 0`, `sig_id 0` can be used to specify a “global” threshold that applies to all rules.
- **sig\_id** specifies the signature ID of an associated rule. `sig_id 0` specifies a “global” filter because it applies to all `sig_ids` for the given `gen_id`.
- **track** defines if the rate is tracked either by source IP address, destination IP address, or by rule. This means that the match statistics are maintained for each unique source IP address, for each unique destination IP address, or they are aggregated at rule level. For rules related to Stream sessions, source and destination means client and server respectively. `track by_rule` and `apply_to` may not be used together.
- **count** specifies the maximum number of rule matches in s seconds before the rate filter limit to is exceeded. c must be nonzero value.
- **seconds** sets the time period over which count is accrued. 0 seconds means count is a total count instead of a specific rate. For example, `rate_filter` may be used to detect if the number of connections to a specific server exceed a specific count. 0 seconds only applies to internal rules (`gen_id 135`) and other use will produce a fatal error by Snort.
- **new\_action** replaces rule action for t seconds. `drop`, `reject`, and `sdrop` can be used only when snort is used in inline mode. `sdrop` and `reject` are conditionally compiled with GIDS.

- **timeout** tells Snort to revert to the original rule action after t seconds. If t is 0, then rule action is never reverted back. An event\_filter may be used to manage number of alerts after the rule action is enabled by rate\_filter.
- **apply\_to** restricts the configuration to only to source or destination IP address (indicated by track parameter) determined by `ip-list`. track by\_rule and apply\_to may not be used together. Note that events are generated during the timeout period, even if the rate falls below the configured limit.

### 3.3.2 event\_filter

Event filtering can be used to reduce the number of logged alerts for noisy rules by limiting the number of times a particular event is logged during a specified time interval. This can be tuned to significantly reduce false alarms. Its format is:

```
event_filter gen_id <gid>, sig_id <sid>, type <limit|threshold|both>, \
    track <by_src|by_dst>, count <c>, seconds <s>
```

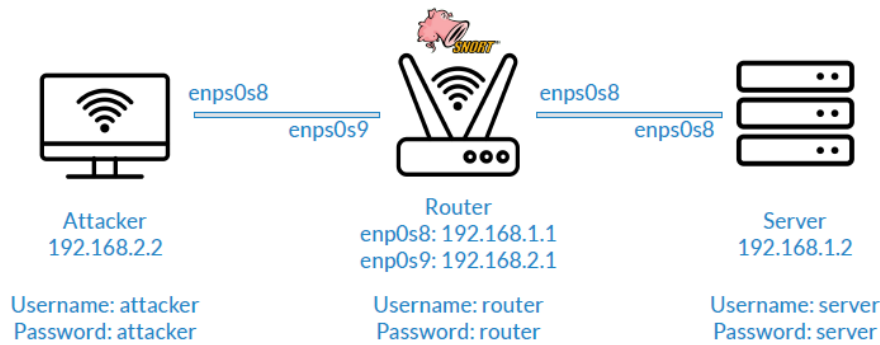
- **gen\_id** specifies the generator ID of an associated rule. gen\_id 0, sig\_id 0 can be used to specify a "global" threshold that applies to all rules.
- **sig\_id** specifies the signature ID of an associated rule. sig\_id 0 specifies a "global" filter because it applies to all sig\_ids for the given gen\_id.
- **type** defines the type of the event filter. Type **limit** alerts on the 1st m events during the time interval, then ignores events for the rest of the time interval. Type **threshold** alerts every m times we see this event during the time interval. Type **both** alerts once per time interval after seeing m occurrences of the event, then ignores any additional events during the time interval.
- **track** specifies if the rate is tracked either by source IP address, or destination IP address. This means count is maintained for each unique source IP addresses, or for each unique destination IP addresses. Ports or anything else are not tracked.
- **count** specifies the number of rule matching in s seconds that will cause event\_filter limit to be exceeded. c must be nonzero value. A value of -1 disables the event filter and can be used to override the global event\_filter.
- **seconds** sets the time period over which count is accrued. s must be nonzero value.

## 4 Environment Setup

For this laboratory, we prepared 3 virtual machines running Ubuntu 20.04; only two of them, the server and the router, have a GUI, in order to reduce the RAM used on host machine to set up the environment. For the same reason we opted to create only these three machine and not also another not malicious client, but for the scope of our exercise our set-up is enough; if you want to verify that the rules block only the malicious traffic coming from the attacker, you can build a client machine following the procedure we used to create the attacker machine.

### 4.1 Network Configuration

The network we created is summarized in the following picture:



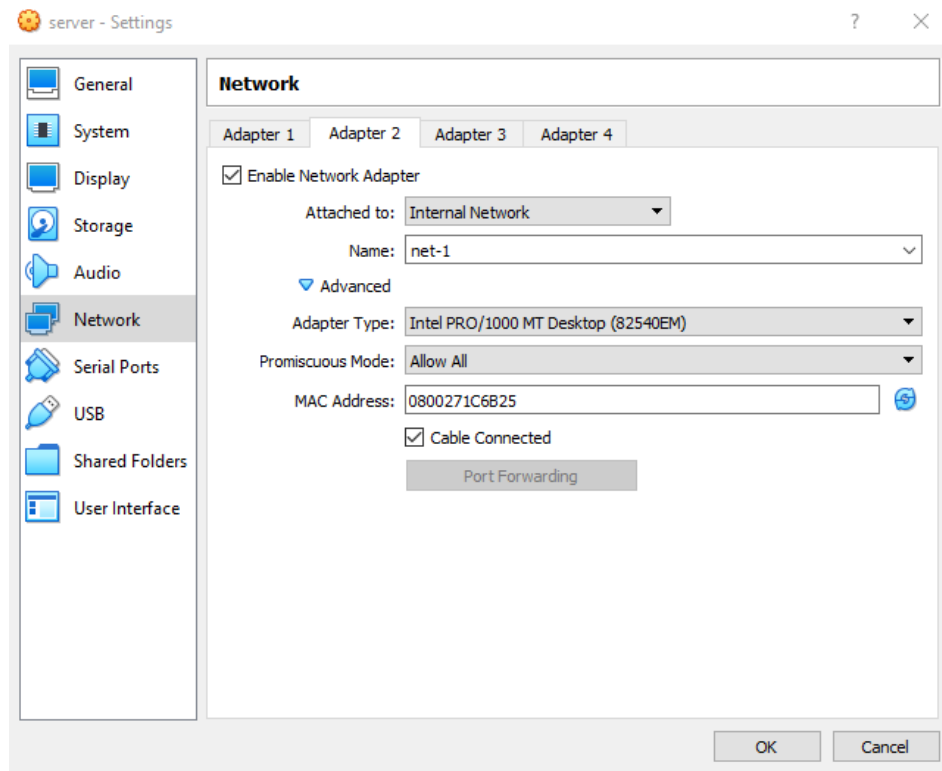
Network Topology

It is possible to notice that we configured two subnets:

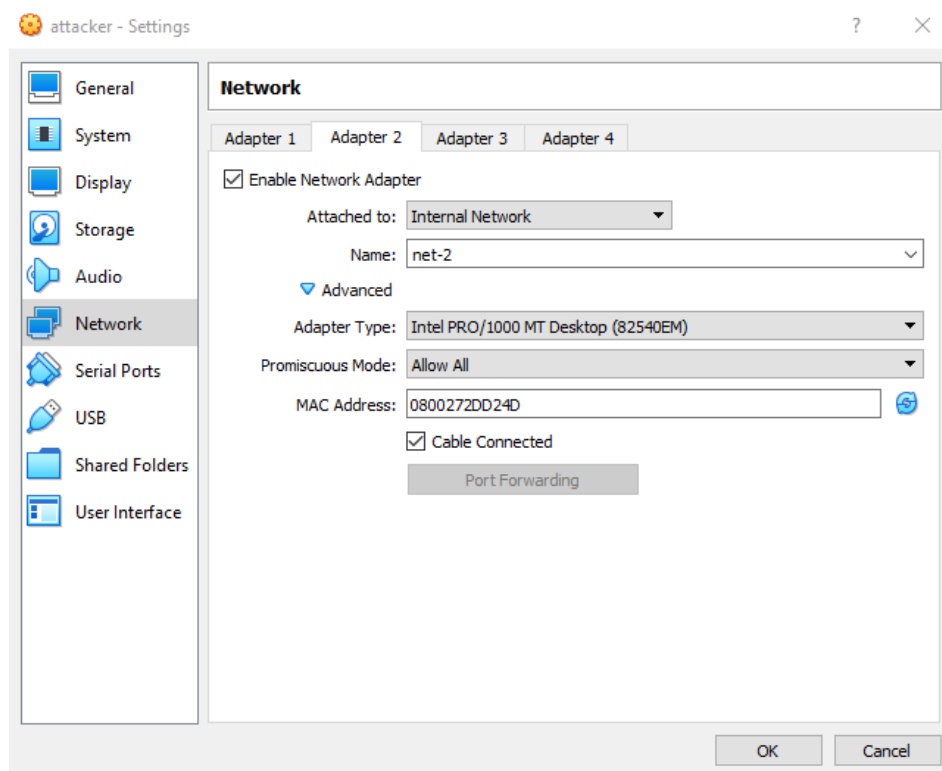
- 192.168.2.0/24, where we placed the attacker machine.
- 192.168.1.0/24, where we put the server.

The router acts as gateway among the two networks. We assigned a static IP address to all four interfaces: the one on the attacker, called enp0s8, has the IP 192.168.2.2, and communicates with the enp0s9 on the router, which has the IP 192.168.2.1. On the other side, the interface enp0s8 on the server, with the IP 192.168.1.2 is connected with the interface enp0s8 on the router, that has the IP 192.168.1.1.

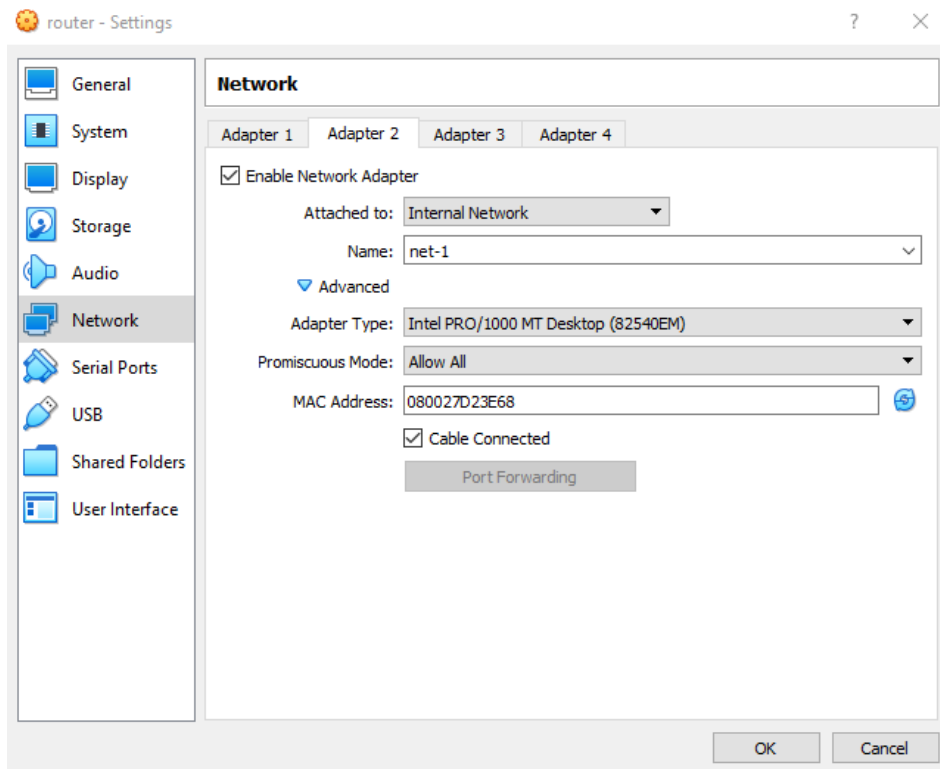
In order to enable the interfaces on the machines, we changed the Virtual Box network settings on every machine, without modifying the first adapter, necessary to connect to Internet, but enabling the second and the third adapters (this latter only on the router) as shown below:



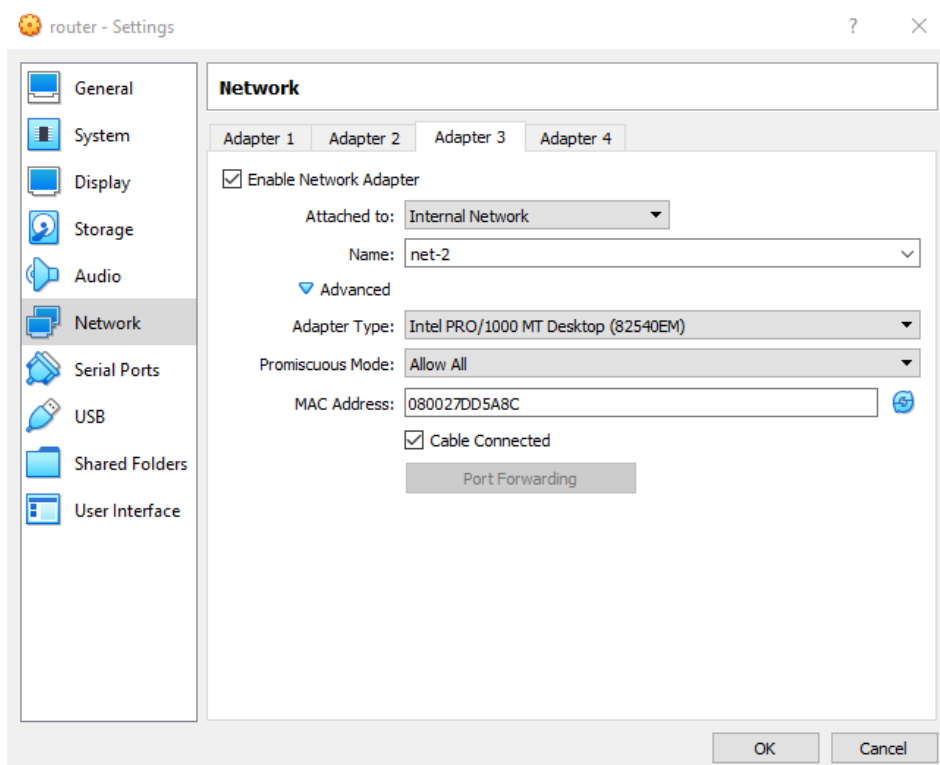
Server Network Settings



Attacker Network Settings



Router Network Settings pt 1



Router Network Settings pt 2

After that, on the router we enabled the port forwarding modifying the `/etc/sysctl.conf` file, using the command: `sudo nano /etc/sysctl.conf`, uncommenting the row: `net.ipv4.ip_forward=1`. Finally, we configured the network interfaces on each machines, modifying the `01-netcfg.yaml` file with the command `sudo nano /etc/netplan/01-netcfg.yaml` as follows:

### Server

```
network:
version: 2
renderer: networkd
ethernets:
  enp0s8:
    dhcp4: no
    addresses: [192.168.1.2/24]
    routes:
      - to: 192.168.2.0/24
        via: 192.168.1.1
```

### Attacker

```
network:
version: 2
renderer: networkd
ethernets:
  enp0s3:
    dhcp4: yes
  enp0s8:
    dhcp4: no
    addresses: [192.168.2.2/24]
    routes:
      - to: 192.168.1.0/24
        via: 192.168.2.1
```

### Router

```
network:
version: 2
renderer: networkd
ethernets:
  enp0s3:
    dhcp4: yes
  enp0s8:
    dhcp4: no
    addresses: [192.168.1.1/24]
  enp0s9:
    dhcp4: no
    addresses: [192.168.2.1/24]
```

To test if the file is written correctly we used the command `sudo netplan try`; since everything was fine, we run `sudo netplan apply`.

## 4.2 Snort Installation

For this laboratory, we need to run Snort in inline mode with the NFQ module, that lets us leverage the power of iptables to make routing decisions. In order to install and configure Snort on the router machine, we followed the procedure described in two guides [1] [2]; their content is summarized and merged below.

First of all we installed the required libraries using this command:

```
sudo apt install -y build-essential gcc libpcrc3-dev zlib1g-dev liblua5.1-dev \
libpcap-dev openssl libssl-dev libnghttp2-dev libdumbnet-dev bison flex libdnet \
autoconf libtool liblzma-dev libnetfilter-queue-dev
```

After that, we created a temporary download folder into the home directory and changed into it with the command

```
mkdir ~/snort_src && cd ~/snort_src
```

At that point, we downloaded the Data Acquisition library (DAQ) (that is used by Snort to make abstract calls to packet capture libraries) from the Snort websites, extracted the source code and jumped into the new directory with the following commands:

```
wget https://www.snort.org/downloads/snort/daq-2.0.7.tar.gz
tar -xvzf daq-2.0.7.tar.gz
cd daq-2.0.7
```

The latest version requires an additional step to auto reconfigure DAQ before running the `config`. We used the command below which requires to have `autoconf` and `libtool` installed:

```
autoreconf -f -i
```

Afterward, we run the configuration script using its default values, then we compiled the program with `make` and finally we installed DAQ:

```
./configure && make && sudo make install
```

Since everything worked well, we were able to see the NFQ DAQ module enabled after having run the `./configure`, as shown below:

```
Build AFPacket DAQ module.. : yes
Build Dump DAQ module..... : yes
Build IPFW DAQ module..... : yes
Build IPQ DAQ module..... : no
Build NFQ DAQ module..... : yes <<<< MUST BE YES
Build PCAP DAQ module..... : yes
Build netmap DAQ module.... :no
```

Hence, we moved back into the temporary folder and installed Snort from the Snort downloads page, extracted the source and changed into the new directory with these commands:

```
cd ~/snort_src
wget https://snort.org/downloads/snort/snort-2.9.17.1.tar.gz
tar -xvzf snort-2.9.17.1.tar.gz
cd snort-2.9.17.1
```

Then we configured the installation with sourcefire enabled, run `make` and `make install`:

```
./configure --enable-sourcefire && make && sudo make install
```

Finally, we updated shared libraries and, since the Snort installation places the Snort binary at `/usr/local/bin/snort`, we created a symlink to `/usr/sbin/snort`:

```
sudo ldconfig
sudo ln -s /usr/local/bin/snort /usr/sbin/snort
```

Snort has been compiled and installed. Now we will configure it.



## 4.3 Snort Configuration

First off, for security reasons we wanted Snort to run as an unprivileged user. We created a snort user and group for this purpose:

```
sudo groupadd snort
sudo useradd snort -r -s /sbin/nologin -c SNORT_IDS -g snort
```

Then we created the folder structure to house the Snort configuration and we set the permissions for the new directories accordingly:

```
sudo mkdir -p /etc/snort/rules
sudo mkdir /var/log/snort
sudo mkdir /usr/local/lib/snort_dynamicrules
sudo chmod -R 5775 /etc/snort
sudo chmod -R 5775 /var/log/snort
sudo chmod -R 5775 /usr/local/lib/snort_dynamicrules
sudo chown -R snort:snort /etc/snort
sudo chown -R snort:snort /var/log/snort
sudo chown -R snort:snort /usr/local/lib/snort_dynamicrules
```

After that, we created new files for the white and black lists as well as the local rules (where we will later write our rules) and copied the configuration files from the download folder:

```
sudo touch /etc/snort/rules/local.rules
sudo touch /etc/snort/rules/white_list.rules
sudo touch /etc/snort/rules/black_list.rules
sudo cp ~/snort_src/snort-2.9.16/etc/*.conf* /etc/snort
sudo cp ~/snort_src/snort-2.9.16/etc/*.map /etc/snort
```

Hence, we edited the `snort.conf` as below:

```
ipvar HOME_NET 192.168.1.0/24           #line 45
ipvar EXTERNAL_NET !$HOME_NET          #line 48
var RULE_PATH /etc/snort/rules          # line 104
var SO_RULE_PATH /etc/snort/so_rules    # line 105
var PREPROC_RULE_PATH /etc/snort/preproc_rules # line 106
var WHITE_LIST_PATH /etc/snort/rules    # line 113
var BLACK_LIST_PATH /etc/snort/rules    # line 114
config daq: nfq                         # line 168
config daq_dir: /usr/local/lib/daq/     # line 169
config daq_mode: inline                 # line 170
config daq_var: queue=4                 # line 171
output unified2: filename snort.log, limit 128 # line 526
include $RULE_PATH/local.rules          # line 551 (uncomment)
```

We also commented all the lines related to site-specific rules, because we didn't download them (we will write our rules during the exercises).

At that point, we added a rule to our iptables that makes every packet that is going to be forwarded (not destined for local delivery) to be scanned by Snort. To do this, the rule moves all forwarded traffic (from the FORWARD queue) to NFQUEUE queue number 4, which is the same queue we specified in our `snort.conf` configuration above, where Snort is listening for (we chose queue 4 arbitrarily). To add this rule to the FORWARD chain, and make it persistent still active after a reboot of the machine, we run as follows:

```
sudo iptables -I FORWARD -j NFQUEUE --queue-num=4
sudo apt install iptables-persistent
sudo systemctl enable netfilter-persistent.service
```

From now on, if Snort is not running, since the system will deliver traffic to that queue, packets will not be processed and will be dropped. Finally we tested the configuration running the following command. The **T** flag indicates we want to test, the **c** flag specifies the `snort.conf` file, and the **Q** flag tells snort that we're working in inline mode (required later for Drop rules to actually drop, instead of just alerting):

```
sudo snort -T -c /etc/snort/snort.conf -Q
```

Since this worked properly, we finished here the laboratory environment setup.

## 5 Exercises

In this chapter we present some exercises that help to understand how Snort works, even though they express only a small part of its power; we ordered them starting from the easier and finishing with the most difficult.

Before you start doing the exercises, you have to:

- On the server:
  - Change the current folder: `/home/server/Desktop/labSnort`
  - Run the flask application: `python3 server.py`
  - Run Wireshark: `sudo wireshark`
- On the router:
  - Start Snort, both as IDS or IPS, unless server and attacker cannot communicate

Remember that you have to write your rules on the `/ect/snort/rules/local.rules` file on the router and, since they are local rules, in order to avoid possible conflict with community rules, they must have `sid > 1000000`. In order to make Snort running with your rules, you have to restart it after having edited the local rules file. All the Snort notions you need are in chapter 3.

## 5.1 PING

Ping is a computer network administration software utility used to test the reachability of a host on an Internet Protocol (IP) network. It is available for virtually all operating systems that have networking capability, including most embedded network administration software. The command to run it is:

```
ping <ip-address>
```

where <ip-address> is the IP address of the machine you want to reach.

### 5.1.1 Exercise

In this first exercise you have to design a rule to detect ping packets coming from outside network and directed to the server.

### 5.1.2 Solution

To complete this exercise, we need to use the `itype` rule option, and the value to detect ping packets is 8. A possible solution could be:

```
alert icmp any any -> $HOME_NET any (itype:8; msg:"PING Detected"; \
sid:1000001; rev:001;)
```

## 5.2 Port Scanning

Port Scanning is a technique used by attackers to probe a server looking for open ports, with the purpose of identifying the network services that are running on the victim and exploit the vulnerabilities.

To execute this technique with a malicious intent an attacker sends a series of requests to a range of server ports addresses on the victim, trying to find an active port.

### 5.2.1 Exercise

In the case of this exercise, we wrote a script that sends a series of SYN packets increasing the port number each time; the purpose of these packets is to start a 3-way handshake and this technique allowed us to know if a port was open, in which case we would receive a SYN-ACK packet as a response, or if it was close, in which case we would receive a TCP-reset packet. The code is available in the appendix B.1. This script was executed on the attacker machine using the command `sudo ./port_scan.sh`.

```
HPING 192.168.1.2 (enp0s8 192.168.1.2): S set, 40 headers + 0 data bytes
len=46 ip=192.168.1.2 ttl=63 DF id=0 sport=4990 flags=RA seq=0 win=0 rtt=4.9 ms
len=46 ip=192.168.1.2 ttl=63 DF id=0 sport=4991 flags=RA seq=1 win=0 rtt=4.5 ms
len=46 ip=192.168.1.2 ttl=63 DF id=0 sport=4992 flags=RA seq=2 win=0 rtt=4.3 ms
len=46 ip=192.168.1.2 ttl=63 DF id=0 sport=4993 flags=RA seq=3 win=0 rtt=3.8 ms
len=46 ip=192.168.1.2 ttl=63 DF id=0 sport=4994 flags=RA seq=4 win=0 rtt=3.8 ms
len=46 ip=192.168.1.2 ttl=63 DF id=0 sport=4995 flags=RA seq=5 win=0 rtt=2.9 ms
len=46 ip=192.168.1.2 ttl=63 DF id=0 sport=4996 flags=RA seq=6 win=0 rtt=2.0 ms
len=46 ip=192.168.1.2 ttl=63 DF id=0 sport=4997 flags=RA seq=7 win=0 rtt=1.2 ms
len=46 ip=192.168.1.2 ttl=63 DF id=0 sport=4998 flags=RA seq=8 win=0 rtt=0.9 ms
len=46 ip=192.168.1.2 ttl=63 DF id=0 sport=4999 flags=RA seq=9 win=0 rtt=8.7 ms
len=46 ip=192.168.1.2 ttl=63 DF id=0 sport=5000 flags=SA seq=10 win=64240 rtt=7.6 ms
len=46 ip=192.168.1.2 ttl=63 DF id=0 sport=5001 flags=RA seq=11 win=0 rtt=7.8 ms
len=46 ip=192.168.1.2 ttl=63 DF id=0 sport=5002 flags=RA seq=12 win=0 rtt=6.9 ms
len=46 ip=192.168.1.2 ttl=63 DF id=0 sport=5003 flags=RA seq=13 win=0 rtt=6.4 ms
len=46 ip=192.168.1.2 ttl=63 DF id=0 sport=5004 flags=RA seq=14 win=0 rtt=5.8 ms
```

Successful Port Scan

We asked the students to write a Snort rule capable of stopping these attacks.

### 5.2.2 Solution

Here we need to write a rule using the flags option and a detection filter. The flags option allows us to specify the kind of packet that we are looking for, in this case S, meaning we are looking for SYN packages.

The detection filter is used because we don't want Snort alerts to every connection attempt, but just only if Snort detect packets coming from the same source a certain number of time in a certain number of seconds, since a few connection attempts should not be considered a port scanning, but just normal traffic.

```
alert tcp !$HOME_NET any -> 192.168.1.2 any (flags:S; msg:"Port Scan detected";
  sid:1000006; rev:001; detection_filter: track by_src, count 5, seconds 60;).
```

This rule is focused on TCP packets coming from outside the server network, from any port and directed to any port, the flag is S used to indicate that we are looking for SYN packets and the detection filter works tracking by source, and activating the rule in case 5 SYN packets from the same source were sent in the span of 60 seconds.

## 5.3 Buffer Overflow

A buffer overflow occurs when the volume of data in input exceeds the capacity of the memory buffer. As a result, the program will write in memory locations adjacent to the buffer, causing memory corruption. Suppose that we are working with a legacy system, that can be reached, using cURL, through an interface on `/buffer` endpoint of our flask server.

The input parameter is called `textsend` and the input form is a `multipart/form-data`

### 5.3.1 Exercise

During the usage of this service, you find out that there is a buffer overflow vulnerability when more than 30 characters are inserted as input (i.e. the buffer length is 30 bytes). When the vulnerability is exploited, the endpoint returns a `Segmentation fault (core dumped)` error.

Given that the attacker uses, as command:

```
curl -X POST -F 'textsend=SomeValue' 192.168.1.2:5000/buffer.
```

try to write a rule that avoid the exploitation of buffer overflow vulnerability

### 5.3.2 Solution

After some test, you can find that, from the cURL command above, Flask receives:

```
[...] name="textsend"\r\n\r\nSomeValue\r\n [...]
```

So, there is the name of the parameter in double quotes. After that, some characters (`\r\n\r\n`) are used as separators, then there is the value and again two characters that say the value is "finished".

We can use two `content` option to write the rule: the first one will match input string until the start of the value, while the second one will tell us if there are more than 30 characters as input:

```
content:"textsend|22 0d 0a 0d 0a|"; content:!"|0d 0a|"; within:30;
```

We mixed plain text with bytetimes:

- 22 is the hexadecimal ascii code for double quotes
- 0d is the hexadecimal ascii code for `\r`
- 0a is the hexadecimal ascii code for `\n`

Thus, the first content will arrange the pointer after the string `\r\n\r\n`. Then, a exploitation of the buffer overflow vulnerability is in progress if we *don't* find `\r\n` in the next 30 characters: this is why we use `within` keyword and the `!` before the value of second `content` option.

The complete snort rule is:

```
alert tcp !$HOME_NET any -> 192.168.1.2 any (msg:\buffer overflow"; sid:1000005;  
content:"textsend|22 0d 0a 0d 0a|"; content:!"|0d 0a|"; within:30;)
```

## 5.4 SQL Injection

A SQL injection occurs when an attacker manipulates the behaviour of a SQL query, leveraging on input mechanism of a web application. This could lead to leak of data to the attacker, as well as corruption of data in the SQL database.

Suppose that you can login to a website through the `/sql` endpoint of the Flask application.

In this endpoint, you insert your `username` and `password` and get back your name and surname. Input fields are part of a form that is `x-www-form-urlencoded`.

The database contains following information in `users` table:

name	surname	username	password
v0	v0	victim0	victim0
v1	v1	victim1	victim1
v2	v2	victim2	victim2
v3	v3	victim3	victim3
v4	v4	victim4	victim4

### 5.4.1 Exercise

An attacker finds out there is an SQL vulnerability on `username` input field.

For example, if value for username field is :

`a' OR 2>1; --.`

the application returns, in the response, all names and surnames of the `users` table.

Given that the attacker uses, as command:

```
curl -d "username=value1" -d "password=value2" 192.168.1.2:5000/sql.
```

try to design a rule to alert when the injection vulnerability is exploited. In particular, try to detect the usage of `OR [number]>[number]` in the `username` field.

### 5.4.2 Solution

After some test, we can find that, from the cURL command above, Flask receives:

`[...] username=value1&password=value2 [...]`

Also, we know that in SQL queries, white spaces of whatever dimension could be inserted between clauses without modifying the overall semantic of the query.

Thus, it could be useful to use `pcre` option in the solution. We need to design a regex to match (for example) `a' OR 2>1`.

We start from the quote symbol: its hexadecimal ASCII number is 27. Before the quote symbol, the attacker could insert whatever alphanumeric character: thus we use `\w`, that can be matched 0 or more times.

After the quote, there could be any number of spaces before the OR. So, we define a character class for white spaces characters: `[\x20\n\t\r\x00]` that could be matched 0 or more times (`\x20` is the hex ASCII code for space character, `\x00` is the null character).

To match numbers, we can use the class of digits `[0-9]` that needs to match at least one time.

The comparison symbol could be:

- equal =
- not equal != or <>
- greater than >
- greater or equal >=

- less than <
- less or equal <=

So we define an alternate match: (|=|<>|>|>=|<|<=)

Putting everything together, we have:

```
pcre: "/username=\w*\x27[\x20\n\t\r\x00]*OR[\x20\n\t\r\x00]*[0-9]+
[\x20\n\t\r\x00]*(|=|<>|>|>=|<|<=) [\x20\n\t\r\x00]*[0-9]+/mi"
```

In the complete rule, we also need to insert a content option, to make a faster match, with value `username=`.

The complete rule is:

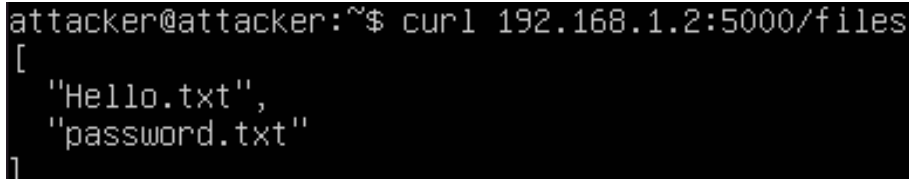
```
alert tcp !$HOME_NET any -> 192.168.1.2 any (msg: "sql injection";
      sid=1000007; rev:001; content: "username=";
pcre: "/username=\w*\x27[\x20\n\t\r\x00]*OR[\x20\n\t\r\x00]*[0-9]+
[\x20\n\t\r\x00]*(|=|<>|>|>=|<|<=) [\x20\n\t\r\x00]*[0-9]+/mi";)
```



## 5.5 Private File Request

If folders are not protected on a server and exposed on the Internet, an attacker can list the files on that server, obtaining sensible information by specifying the name of the file; this vulnerability is called exposure information. In the case of our exercise the port 5000 was left open on purpose, and it is possible to access the exposed folder through the following request:

```
curl 192.168.1.2:5000/files.
```

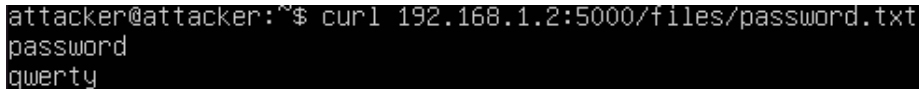
A terminal window with a black background. The prompt is 'attacker@attacker:~\$'. The command entered is 'curl 192.168.1.2:5000/files'. The output is a JSON array: ['Hello.txt', 'password.txt'].

```
attacker@attacker:~$ curl 192.168.1.2:5000/files
[
  "Hello.txt",
  "password.txt"
]
```

File List

This allows the attacker to see the file list and to access the information contained in them with a specific request:

```
curl 192.168.1.2:5000/files/password.txt.
```

A terminal window with a black background. The prompt is 'attacker@attacker:~\$'. The command entered is 'curl 192.168.1.2:5000/files/password.txt'. The output is the text 'password' followed by 'qwerty' on the next line.

```
attacker@attacker:~$ curl 192.168.1.2:5000/files/password.txt
password
qwerty
```

Sensitive Information Exposure

### 5.5.1 Exercise

The exercise we prepared asked to create a rule that prevents the attacker from reading the content of the file *password.txt*, that we want to protect due to the sensitive information contained in it, while we were not interested in protecting the file *Hello.txt*.

### 5.5.2 Solution

A possible rule to solve this exercise should be written utilizing the *content* parameter, that gives the user the possibility to specify the exact content that we want to set the rule for.

```
drop tcp !$HOME_NET any -> 192.168.1.2 any (msg:"Private File Read Attempt Detected";
      sid:1000003; content:"password");).
```

## 5.6 SYN Flood Attack

This is a common and easy DoS attack, that consist in overwhelming the target's resources by sending a high volume of SYN packets to the server using spoofed IP addresses causing the server to send a reply (SYN-ACK) and leave its ports half-open, awaiting for a reply from a host that does not exist. In this state, the target struggles to handle traffic which in turn will increase CPU usage and memory consumption ultimately leading to the exhaustion of its resources (CPU and RAM). At this point the server will no longer be able to serve legitimate client requests and ultimately lead to a Denial-of-Service. We create a script that automate this attack, and you can find a description of in the appendix B.2. To run it, simply wrote `sudo ./flooder.sh` on the attacker machine.

### 5.6.1 Exercise

In this last exercise it is required to design a rule to protect the server against SYN Flood Attacks.

### 5.6.2 Solution

To complete this exercise, we need to use both the `event_filter` and the `rate_filter`, the former because the high number of alerts Snort will print due to the high volume of packets generated by the script, the latter because we want to start dropping packets not from the beginning, otherwise we would be blocking everybody from connecting to the server. So we need to create three things:

- the `rate_filter`
- the `event_filter`
- the rules that alert when a SYN packet is directed to the server open port

Hence, a possible solution could be:

```
rate_filter gen_id 1, sig_id 1000004, track by_dst, count 100, seconds 1, \
    new_action drop, timeout 5
event_filter gen_id 1, sig_id 1000004, track by_dst, count 70, seconds 1, \
    type threshold
alert tcp any any -> 192.168.1.2 5000 (flags:S; msg: "Possible DoS Attack"; \
    sid:1000004;)
```

# Code

Here we reported all the code we wrote preparing this laboratory.

## A Server

On server, we run a python script which implemented a flask web application. The code is the following:

### A.1 server.py

```
from flask import Flask, render_template, redirect, url_for, request,
from flask import session, abort, jsonify, send_from_directory
import string, random, subprocess, os, sqlite3

DIR = "/home/server/Desktop/labSnort/files"

app = Flask(__name__)
chars=string.ascii_letters + string.digits
app.secret_key = ''.join(random.choice(chars) for i in range(25))

@app.route('/sql', methods=['GET', 'POST'])
def login():
    error = None
    if request.method == 'POST':
        insertedUser=request.form['username']
        insertedPassword=request.form['password']
        con = sqlite3.connect('example.db')
        cur = con.cursor()
        print(insertedUser)
        print(insertedPassword)
        cur.execute("SELECT name, surname FROM users WHERE username = \
            '%s' and password = '%s'" % (insertedUser, insertedPassword))
        rows = cur.fetchall()
        returnObject={}
        returnObject['fetched']=rows
        return returnObject
    return render_template('sqlLogin.html', error=error)

@app.route('/buffer', methods=['GET', 'POST'])
def home():
    error = None
    if request.method == 'POST':
        textInserted=request.form['textsend']
        result = subprocess.run(['./exploitable', textInserted],
                                stdout=subprocess.PIPE, stderr=subprocess.PIPE)
        returnedObject = {}
        returnedObject['code'] = -1*result.returncode
```

```

        if result.returncode<0:
            returnedObject['output']="Segmentation Fault (core dumped)"
        else:
            returnedObject['output']=result.stdout.decode('utf-8')
        return returnedObject
    return render_template('buffer.html', error=error)

@app.route('/files', methods=['GET'])
def list_files():
    files = []
    for filename in os.listdir(DIR):
        path = os.path.join(DIR, filename)
        if os.path.isfile(path):
            files.append(filename)
    return jsonify(files)

@app.route('/files/<path:path>')
def get_file(path):
    return send_from_directory(DIR, path, as_attachment=True)

if __name__ == '__main__':
    app.run(debug=True, host='192.168.1.2', port=5000)

```

## A.2 sourceC.cc

The flask application called an executable, compiled with g++ and flags `-m32 -fno-stack-protector` from the following source code:

```

#include <iostream>
#include <cstring>
#include <cstdlib>
#include <unistd.h>

using namespace std;
void printResult(char * input){
    char buffer[30];
    strcpy(buffer, input);
}

int main(int argc, char *argv[]){
    if (argc < 2){
        printf("strcpy() NOT executed...\n");
        printf("Syntax: %s <characters>\n", argv[0]);
        exit(0);
    }

    printResult(argv[1]);
    printf("function correctly executed...\n");
    return 0;
}

```

## B Attacker

Here we put the scripts we wrote for the attacker. Both use **hping3**, a popular TCP penetration testing tool; we installed it on the attacker machine using:

```
sudo apt install hping3
```

### B.1 port.scan.sh

This is the script we created to implement the port scanning.

```
#!/bin/bash

echo "Port Scan Started"
hping3 -S 192.168.1.2 -p ++4990 & export
PID=$!
sleep 15
kill $PID
sleep 1
echo "Port Scan Stopped"
```

We are sending packets specifying that the SYN Flag (-S) should be enabled. To detect open ports on our victim's server, we start sending packets to port 4990 onwards (-p ++4990). The attack last 15 seconds and is then automatically stopped.

### B.2 flooder.sh

This is the script we created to implement the SYN Flood Attack.

```
#!/bin/bash

echo "Flooder Started"
hping3 -d 120 -S -p 5000 --flood --rand-source 192.168.1.2 & export
FLOODER_PID=$!
sleep 10
kill $FLOODER_PID
sleep 0.5
echo "Flooder Stopped"
```

We are sending packets of 120 bytes (-d 120) each, specifying that the SYN Flag (-S) should be enabled. To direct the attack to our victim's server we specify port 5000 (-p 5000) and use the `--flood` flag to send packets as fast as possible. The `--rand-source` flag generates spoofed IP addresses to disguise the real source and avoid detection but at the same time stop the victim's SYN-ACK reply packets from reaching the attacker.

The attack last 10 seconds and is then automatically stopped.

# References

- [1] *Snort Installation Guide*. <https://upcloud.com/community/tutorials/install-snort-ubuntu>.
- [2] *Snort IPS Configuration Guide*. <http://sublimerobots.com/2017/06/snort-ips-with-nfq-routing-on-ubuntu>.
- [3] *Snort Manual*. <https://www.snort.org/documents/snort-users-manual>.