UNIVERSITY OF TRENTO

NETWORK SECURITY REPORT

Key Reinstallation Attack

Stefano Di Santo

Alessandra Morellini

Tommaso Sacchetti

2020/2021

Contents

1 Introduction **3**

1.1 High-level explanation of 4-way handshake 3

2 Laboratory **4**

2.1 Custom simulation 4

2.2 Video demonstration 7

2.2.1 Prerequisites 7

2.2.2 Before every usage 8

2.2.3 The test 8

2.3 Mininet-WiFi 10

2.3.1 Simulation on a patched system 10

2.3.2 Simulation on a vulnerable system 16

2.4 Packet captures 17

1 Introduction

The Key Reinstallation Attack (KRACK) was discovered in 2017 by Mathy Vanhoef, a researcher of Imec-DistriNet at KU Leuvenis. This attack exploit a vulnerability of the WiFi 802.11 standard, in particular of the 4-way handshake. It affects the Wi-Fi standard itself, and can be used to read sensitive information that was previously assumed to be safely encrypted. Considering that the vulnerability is on the protocol itself , and not on individual products or implementations, the majority of them was vulnerable. As soon as the exploitation was made public, all the major vendors provide their devices with security patches.

1.1 High-level explanation of 4-way handshake

The WPA2 4-way handshake provides mutual authentication between a supplicant and an authenticator based on a shared secret (Pairwise Master Key), which is derived from the password that enables users to connect to the network.
The handshake also negotiates the session key (PTK) which is derived from a combination of the PMK, the MAC addresses of both the supplicant and authenticator and two nonces: the Authenticator Nonce (ANonce) and Supplicant Nonce (SNonce).
In KRACK attack, the adversary trick the victim to reinstall an already-in-use key manipulating the handshake messages. This cause that the associated incremental values and received packet numbers are reset to their initial value. To guarantee security a key should never be used twice but due to the design of 802.11 standard the WPA2 protocol does not guarantee this property.

2 Laboratory

All the material used during laboratory lesson is available on a github repository [1], while the video showed is available on youtube [2]. Beware that the video has been recorded outside a virtual environment, instruction on how to reproduce it will follow.
In order to replicate what we saw during the lab it is necessary to clone the github repository containing all the files. From the provided Virtual Machine (VM) we should execute the `envsetup` script, which is located in the home folder.

```
    ./envsetup.sh
```

Once completed we can move inside the `network-security-2021` folder.

```
    cd network-security-2021
```

This folder contains all the files needed to replicate the laboratory.

2.1 Custom simulation

This simulation is written in Golang and compiled to native binaries. It is an attempt to replicate the WPA2 handshake logic using TCP sockets.
From the main folder (`network-security-2021`), enter the `simulation` sub-folder:

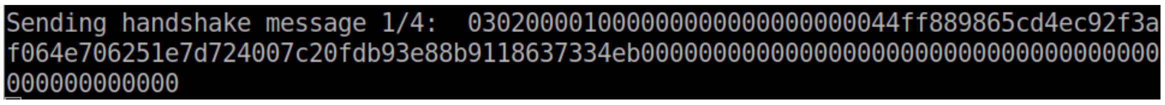
```
    cd simulation
```

Here is contained a script which will compile the sources (`client.go`, `mitm.go` and `ap.go`) into three executable.The script can be launched via:

```
    ./start.sh
```

The three terminals should now be opened with their own name: AP, Client and MitM. The simulation requires user interaction, to start it we can move to the AP terminal.

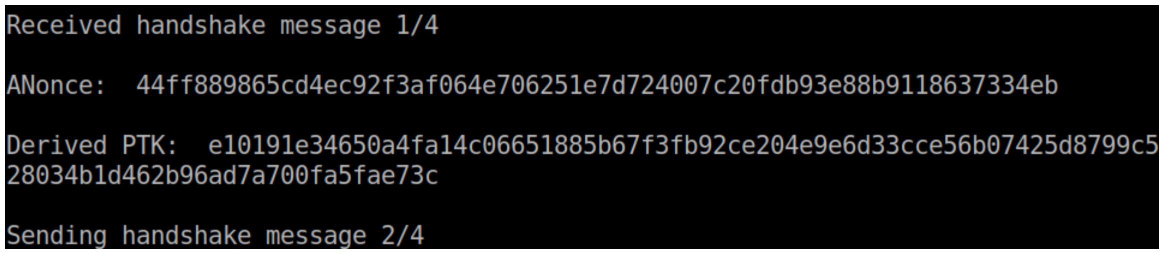
The AP will send the first handshake message (1/4) to the client containing `PacketType = 3`, `DescriptorType = 1`, `KeyLength = 16`, `Counter = 0` and a 32-byte length `ANonce`.



```
Sending handshake message 1/4:  030200001000000000000000000044ff889865cd4ec92f3a
f064e706251e7d724007c20fdb93e88b9118637334eb000000000000000000000000000000000000
000000000000
```

Figure 1: First handshake message AP -> Client

The client upon receipt of the first message will derive the Pairwise Temporal Key (PTK) and send back the second handshake message (2/4) containing its SNonce.

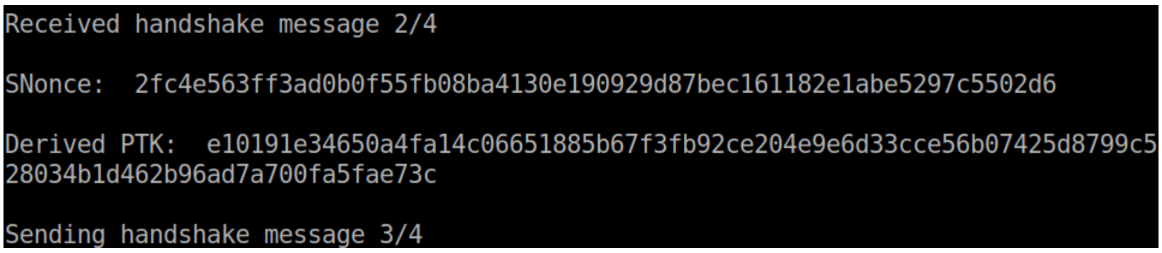


```
Received handshake message 1/4

ANonce:  44ff889865cd4ec92f3af064e706251e7d724007c20fdb93e88b9118637334eb

Derived PTK:  e10191e34650a4fa14c06651885b67f3fb92ce204e9e6d33cce56b07425d8799c5
28034b1d462b96ad7a700fa5fae73c

Sending handshake message 2/4
```

Figure 2: Cient PTK derivation

[1]Beta branch: `https://github.com/sacca97/network-security-2021`
[2]`https://youtu.be/Wv_VFHK5RLQ`

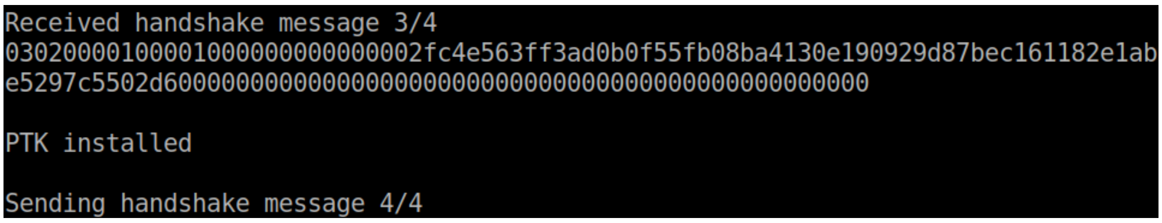
Receiving the SNonce, the AP will derive the PTK and install it, then it will send back a third handshake message (3/4) to confirm the correct installation of the PTK.



```
Received handshake message 2/4

SNonce:  2fc4e563ff3ad0b0f55fb08ba4130e190929d87bec161182e1abe5297c5502d6

Derived PTK:  e10191e34650a4fa14c06651885b67f3fb92ce204e9e6d33cce56b07425d8799c5
28034b1d462b96ad7a700fa5fae73c

Sending handshake message 3/4
```

Figure 3: AP PTK derivation

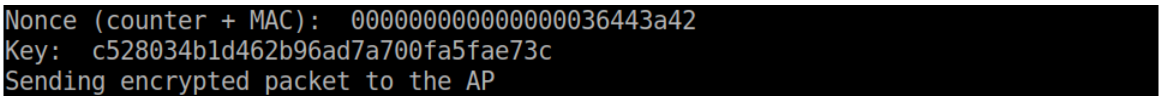
When receiving the third handshake message (3/4), the client will reply with a fourth handshake message (4/4), after sending this message it will install the PTK and reset the counter.



```
Received handshake message 3/4
03020000010000100000000000002fc4e563ff3ad0b0f55fb08ba4130e190929d87bec161182e1ab
e5297c5502d6000000000000000000000000000000000000000000000000

PTK installed

Sending handshake message 4/4
```

Figure 4: Client PTK installation

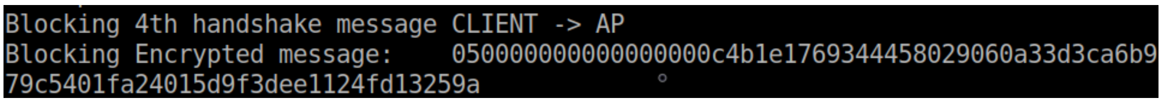
In this simulation the client immediately sends an encrypted packet, so it's possible to note the nonce and the key used to encrypt this packet with AES-GCMP.



```
Nonce (counter + MAC):  000000000000000036443a42
Key:  c528034b1d462b96ad7a700fa5fae73c
Sending encrypted packet to the AP
```

Figure 5: Client encrypted packet

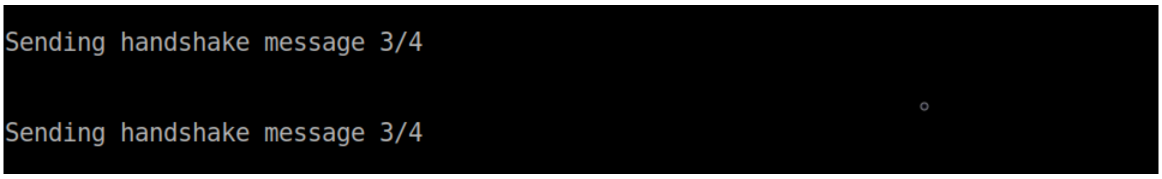
The Man-in-the-Middle will block handshake message 4 and any encrypted message sent after that.



```
Blocking 4th handshake message CLIENT -> AP
Blocking Encrypted message:    050000000000000000c4b1e1769344458029060a33d3ca6b9
79c5401fa24015d9f3dee1124fd13259a
```

Figure 6: MitM blocking messages

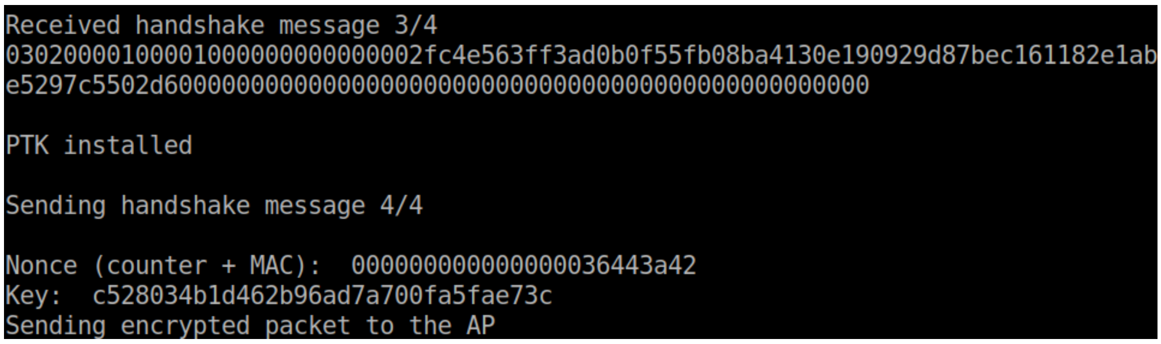
The access point will send back another handshake message 3/4, since it never received the last handshake message.



```
Sending handshake message 3/4


Sending handshake message 3/4
```

Figure 7: Second message 3 from AP

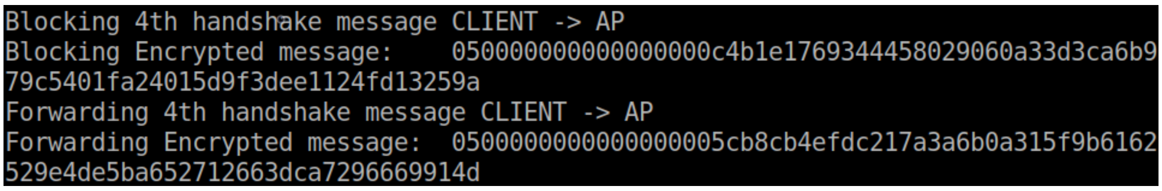
Receiving this message will cause the client to re-install the PTK and re-send the message 4, thereby resetting the counter used to generate the encryption nonce. In fact, it is possible to see in Figure 8 that for encrypting the message, the client is using the same key and the same nonce as in Figure 5.



```
Received handshake message 3/4
03020000010000100000000000000002fc4e563ff3ad0b0f55fb08ba4130e190929d87bec161182e1ab
e5297c5502d6000000000000000000000000000000000000000000000000

PTK installed

Sending handshake message 4/4

Nonce (counter + MAC):  000000000000000036443a42
Key:  c528034b1d462b96ad7a700fa5fae73c
Sending encrypted packet to the AP
```

Figure 8: New client encrypted packet

In the MitM terminal (Figure 9) it is possible to see the old blocked messages and the new ones sent from the client to the AP. Furthermore it is possible to notice how the two counters are identical, since the combination of the counter and the sender MAC address is used to derive the Initialization Vector (IV) for the AES cipher. Those are two different packets encrypted with the same key and the same nonce.



```
Blocking 4th handshake message CLIENT -> AP
Blocking Encrypted message:    050000000000000000c4b1e1769344458029060a33d3ca6b9
79c5401fa24015d9f3dee1124fd13259a
Forwarding 4th handshake message CLIENT -> AP
Forwarding Encrypted message:  0500000000000000005cb8cb4efdc217a3a6b0a315f9b6162
529e4de5ba652712663dca7296669914d
```

Figure 9: MitM logs

To exit the simulation simply press crtl + c when in the MitM terminal, then close everything.

2.2 Video demonstration

2.2.1 Prerequisites

To replicate the test there are various alternatives. It is important to specify that the test is based on the idea of the attack but it is not the attack itself. Indeed, the implementation creates a fictitious Access Point to which the client connects, then the AP will force the unintended behaviour of the client in order to see if the client is vulnerable or not.
To perform this test we need to setup one of the following:

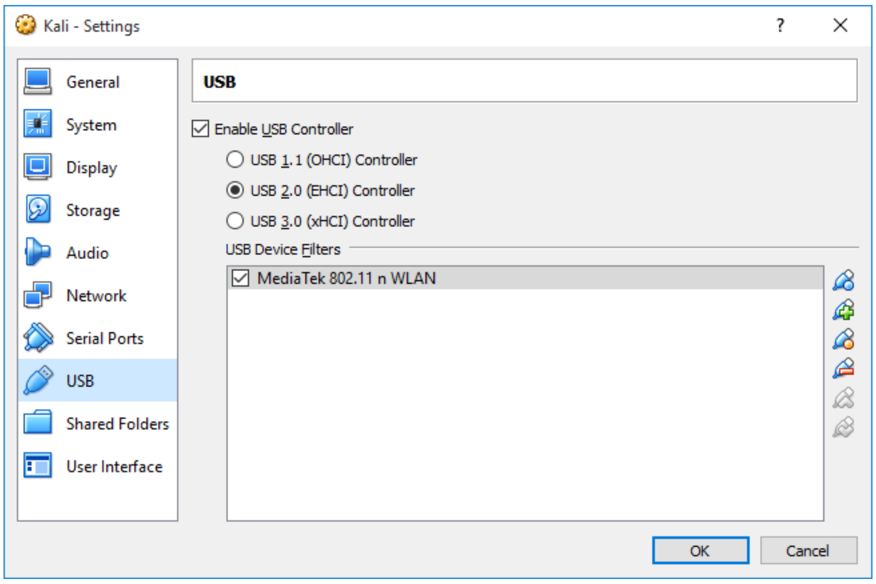
- PC with Kali Linux (live or installed) and a WiFi NIC
- Virtual Machine with Kali Linux and a USB Wi-Fi NIC that supports monitor and AP mode.

If using directly Kali Linux, no further configuration is needed and we can skip the VM setup.

If using a Kali based VM we need to connect a USB Wi-Fi NIC in order to have direct access to the Wi-Fi from inside the VM.
To setup the adapter simply follow the next next steps:

1. Download and install the Extension Pack to enable USB port from `https://www.virtualbox.org/wiki/Downloads`
2. Connect the adapter
3. Enable USB 2.0 in virtual box settings
4. Select the adapter
5. Save configuration and reboot the VM

Now the USB settings of Virtual Box should look like this (the name of the Wireless NIC can change).



USB

[✓] Enable USB Controller

USB 1.1 (OHCI) Controller [✓ USB 2.0 (EHCI) Controller] USB 3.0 (xHCI) Controller

USB Device Filters

[✓] MediaTek 802.11 n WLAN

Figure 10: VM Configuration

After this, we can start the VM and prepare it to execute our test. Fist of all download all the required files from the github repository with the command:

```
git clone https://github.com/vanhoefm/krackattacks-scripts.git
```

To install the required dependencies (on Kali), execute:

```
sudo apt update
sudo apt install libnl-3-dev libnl-genl-3-dev pkg-config libssl-dev
   net-tools git sysfsutils virtualenv
```

Then we must disable hardware encryption (it is recommended to reboot after this step):

```
cd krackattack
sudo ./disable-hwcrypto.sh
```

If needed hardware encryption can be later re-enabled using the script `reenable-hwcrypto.sh`.
Now we have to compile the modified hostapd executable, for this we can simply run the provided script:

```
cd krackattack
./build.sh
```

To avoid errors when importing the python modules in the test script, we will now create a virtual environment with the dependencies (listed in `krackattack/requirements.txt`), this can be done using the provided script.

```
./pysetup.sh
```

2.2.2 Before every usage

Every time before using the scripts we must disable the Wi-Fi in the network manager and then allow our script to still access it, this can be done from the main folder by executing:

```
nmcli radio wifi off
sudo rfkill unblock wifi
cd krackattack
sudo su
source venv/bin/activate
```

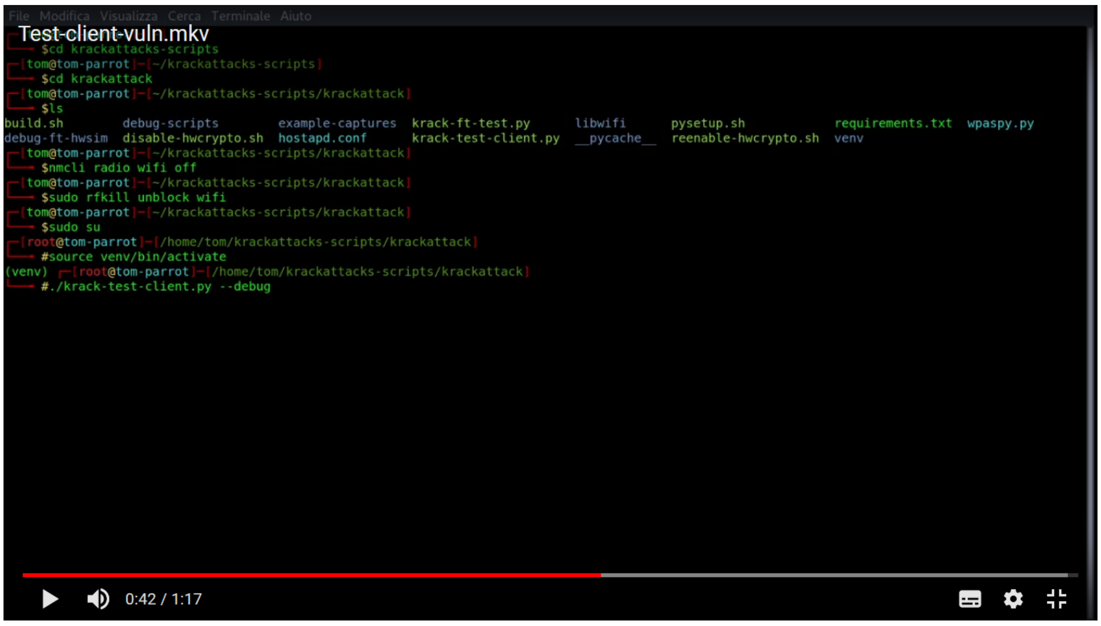
With the last two commands we also activated the virtual environment with root privileges, now we can execute the scripts without repeating the previous steps as long as the terminal is open.

2.2.3 The test

First modify `hostapd/hostapd.conf` and edit the line `interface=` to specify the Wi-Fi interface that will be used to execute the tests. Note that for all tests, once the script is running, the device being tested must connect to the SSID **testnetwork** using the password **abcdefgh**. Settings of the AP can be changed by modifying `hostapd/hostapd.conf`. In all tests the client must use DHCP to get an IP after connecting to the Wi-Fi network. This is because some tests only start after the client has requested an IP using DHCP.
It is possible now to run the following tests located in the krackattacks/ directory typing:

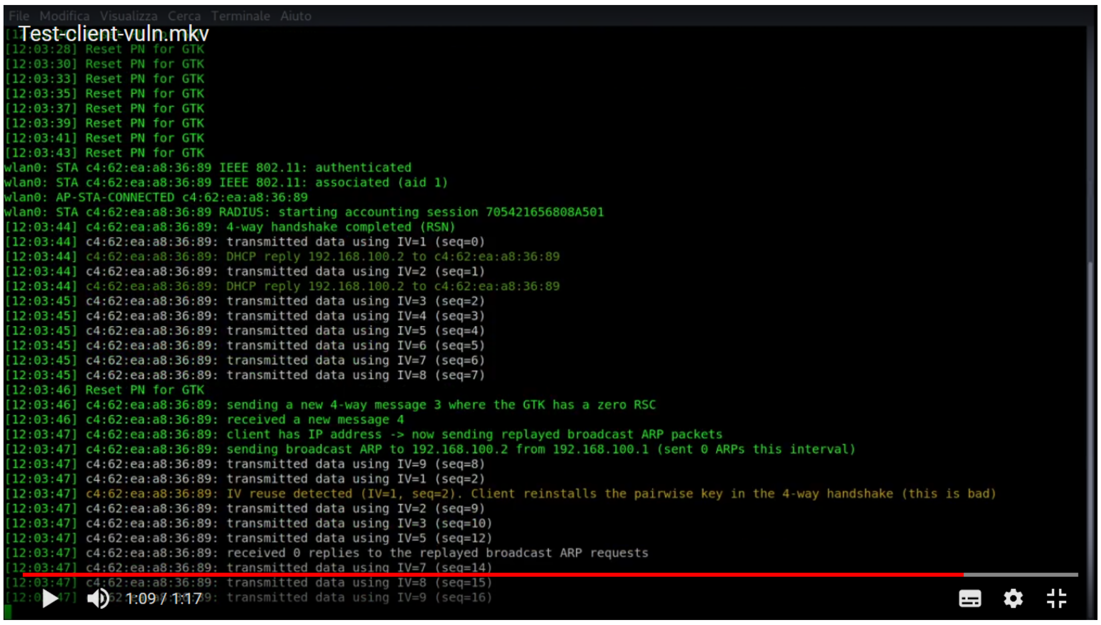
```
./krack-test-client.py
```

This tests the key reinstallations attack on the 4-way handshake by sending repeatedly encrypted message 3's to the client. In other words, this tests for CVE-2017-13077 (the vulnerability with the highest impact) and for CVE-2017-13078. The script monitors the traffic sent by the client to see if the pairwise key is being reinstalled. Note that this effectively performs two tests: whether the pairwise key is reinstalled, and whether the group key is reinstalled. Make sure the client requests an IP using DHCP for the group key reinstallation test to start. To assure the client is sending enough unicast frames, you can optionally ping the AP: ping 192.168.100.254.



```
└──╼ $cd krackattacks-scripts
┌─[tom@tom-parrot]─[~/krackattacks-scripts]
└──╼ $cd krackattack
┌─[tom@tom-parrot]─[~/krackattacks-scripts/krackattack]
└──╼ $ls
build.sh        debug-scripts        example-captures  krack-ft-test.py      libwifi      pysetup.sh            requirements.txt  wpaspy.py
debug-ft-hwsim  disable-hwcrypto.sh  hostapd.conf      krack-test-client.py  __pycache__  reenable-hwcrypto.sh  venv
┌─[tom@tom-parrot]─[~/krackattacks-scripts/krackattack]
└──╼ $nmcli radio wifi off
┌─[tom@tom-parrot]─[~/krackattacks-scripts/krackattack]
└──╼ $sudo rfkill unblock wifi
┌─[tom@tom-parrot]─[~/krackattacks-scripts/krackattack]
└──╼ $sudo su
┌─[root@tom-parrot]─[/home/tom/krackattacks-scripts/krackattack]
└──╼ #source venv/bin/activate
(venv) ┌─[root@tom-parrot]─[/home/tom/krackattacks-scripts/krackattack]
└──╼ #./krack-test-client.py --debug
```

Figure 11: Screenshot of the script behaviour 1



```
[12:03:28] Reset PN for GTK
[12:03:30] Reset PN for GTK
[12:03:33] Reset PN for GTK
[12:03:35] Reset PN for GTK
[12:03:37] Reset PN for GTK
[12:03:39] Reset PN for GTK
[12:03:41] Reset PN for GTK
[12:03:43] Reset PN for GTK
wlan0: STA c4:62:ea:a8:36:89 IEEE 802.11: authenticated
wlan0: STA c4:62:ea:a8:36:89 IEEE 802.11: associated (aid 1)
wlan0: AP-STA-CONNECTED c4:62:ea:a8:36:89
wlan0: STA c4:62:ea:a8:36:89 RADIUS: starting accounting session 705421656808A501
[12:03:44] c4:62:ea:a8:36:89: 4-way handshake completed (RSN)
[12:03:44] c4:62:ea:a8:36:89: transmitted data using IV=1 (seq=0)
[12:03:44] c4:62:ea:a8:36:89: DHCP reply 192.168.100.2 to c4:62:ea:a8:36:89
[12:03:44] c4:62:ea:a8:36:89: transmitted data using IV=2 (seq=1)
[12:03:44] c4:62:ea:a8:36:89: DHCP reply 192.168.100.2 to c4:62:ea:a8:36:89
[12:03:45] c4:62:ea:a8:36:89: transmitted data using IV=3 (seq=2)
[12:03:45] c4:62:ea:a8:36:89: transmitted data using IV=4 (seq=3)
[12:03:45] c4:62:ea:a8:36:89: transmitted data using IV=5 (seq=4)
[12:03:45] c4:62:ea:a8:36:89: transmitted data using IV=6 (seq=5)
[12:03:45] c4:62:ea:a8:36:89: transmitted data using IV=7 (seq=6)
[12:03:45] c4:62:ea:a8:36:89: transmitted data using IV=8 (seq=7)
[12:03:46] Reset PN for GTK
[12:03:46] c4:62:ea:a8:36:89: sending a new 4-way message 3 where the GTK has a zero RSC
[12:03:46] c4:62:ea:a8:36:89: received a new message 4
[12:03:47] c4:62:ea:a8:36:89: client has IP address -> now sending replayed broadcast ARP packets
[12:03:47] c4:62:ea:a8:36:89: sending broadcast ARP to 192.168.100.2 from 192.168.100.1 (sent 0 ARPs this interval)
[12:03:47] c4:62:ea:a8:36:89: transmitted data using IV=9 (seq=8)
[12:03:47] c4:62:ea:a8:36:89: transmitted data using IV=1 (seq=2)
[12:03:47] c4:62:ea:a8:36:89: IV reuse detected (IV=1, seq=2). Client reinstalls the pairwise key in the 4-way handshake (this is bad)
[12:03:47] c4:62:ea:a8:36:89: transmitted data using IV=2 (seq=9)
[12:03:47] c4:62:ea:a8:36:89: transmitted data using IV=3 (seq=10)
[12:03:47] c4:62:ea:a8:36:89: transmitted data using IV=5 (seq=12)
[12:03:47] c4:62:ea:a8:36:89: received 0 replies to the replayed broadcast ARP requests
[12:03:47] c4:62:ea:a8:36:89: transmitted data using IV=7 (seq=14)
[12:03:47] c4:62:ea:a8:36:89: transmitted data using IV=8 (seq=15)
[12:03:47] c4:62:ea:a8:36:89: transmitted data using IV=9 (seq=16)
```

Figure 12: Screenshot of the script behaviour 2

It is important to to perform these tests in a room with little interference. A high amount of packet loss will make this script less reliable!
Additionally, we can add the `-debug` parameter for more debugging output.
Finally, if everything has been done well, we should see (if the device is vulnerable) that the device has reused the IV for some packet.

2.3 Mininet-WiFi

Mininet-WiFi is a fork of the Mininet SDN network emulator, it extend the functionality of Mininet by adding virtualized WiFi devices based on the standard Linux wireless drivers and the 80211_hwsim wireless simulation driver. The platform was used to replicate a virtual environment with two access points (APs) and one station (STA), the goal of the simulation was to test a fully patched system for the fast BSS transition handshake vulnerability.

2.3.1 Simulation on a patched system

From the main folder, we move to the mininet directory. In this folder there is everything needed for our simulation.

```
cd network-security-2021/mininet
```

The file `minisetup.py` contains the network configuration scripts for the configuration of the network, the script `pysetup.sh` is the same metioned in subsubsection 2.2.1 and it is needed to create the virtual environment with all the requirement needed to execute the test. The file `krack-ft-test.py` is the actual test script will be executed by station 1 to "attack" one of the access point.

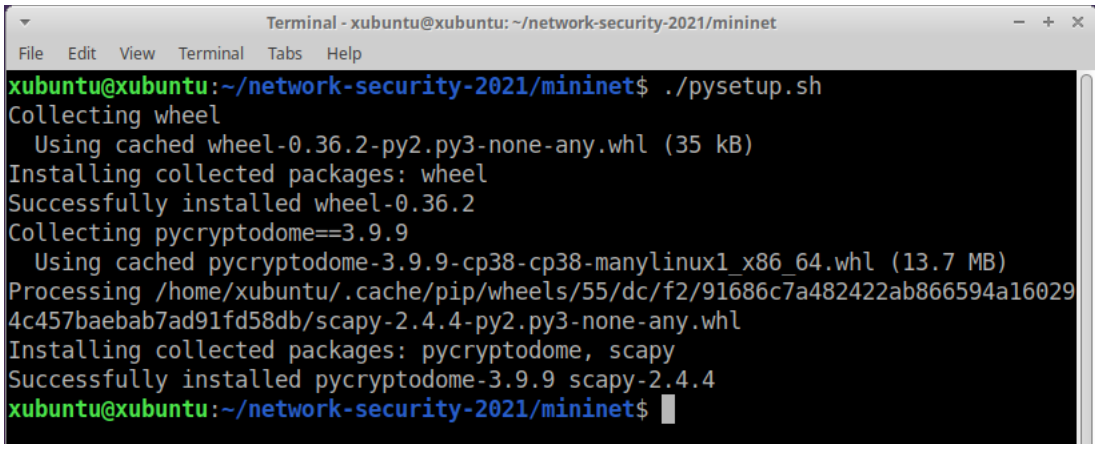
Listing 1: minisetup.py

```python
#!/usr/bin/python3

from time import sleep

from mininet.log import setLogLevel, info
from mininet.term import makeTerm
from mn_wifi.net import MininetWithControlWNet, Mininet_wifi
from mn_wifi.cli import CLI
from mn_wifi.link import wmediumd
from mn_wifi.wmediumdConnector import interference


def topology():

    "Create a network."
    net = Mininet_wifi(link=wmediumd, wmediumd_mode=interference)

    info("*** Creating nodes\n")
    sta1 = net.addStation('sta1', ip='10.0.0.1/8', position='50,0,0',
                          encrypt='wpa2')
    ap1 = net.addStation('ap1', mac='02:00:00:00:01:00',
                         ip='10.0.0.101/8', position='10,30,0')
    ap2 = net.addStation('ap2', mac='02:00:00:00:02:00',
                         ip='10.0.0.102/8', position='100,30,0')

    info("*** Configuring Propagation Model\n")
    net.setPropagationModel(model="logDistance", exp=3.5)

    info("*** Configuring wifi nodes\n")
    net.configureWifiNodes()

    ap1.setMasterMode(intf='ap1-wlan0', ssid='handover', channel='1',
                      ieee80211r=True, mobility_domain='a1b2',
                        passwd='123456789a', encrypt='wpa2')
    ap2.setMasterMode(intf='ap2-wlan0', ssid='handover', channel='6',
                        ieee80211r=True, mobility_domain='a1b2',
                        passwd='123456789a', encrypt='wpa2')

    info("*** Plotting Graph\n")
    net.plotGraph(min_x=-100, min_y=-100, max_x=200, max_y=200)

    info("*** Starting network\n")
    net.build()

    sta1.cmd("iw dev sta1-wlan0 interface add mon0 type monitor")
    sta1.cmd("ifconfig mon0 up")

    sleep(5)
    makeTerm(sta1, title='Sta1', cmd="bash -c 'echo \"AP Scanning\"
        && iw dev sta1-wlan0 scan; bash'")

    info("*** Running CLI\n")
    CLI(net)

    info("*** Stopping network\n")
    net.stop()


if __name__ == '__main__':
    setLogLevel('info')
    topology()
```

First of all, from the mininet folder, we have to run the `pysetup.sh` which will create the python3 virtual environment with the required dependencies to execute the test script, details are into the `requirements.txt` file.

```
sudo ./pysetup.sh
```

The previous command should give the following output:


Terminal - xubuntu@xubuntu: ~/network-security-2021/mininet
File Edit View Terminal Tabs Help

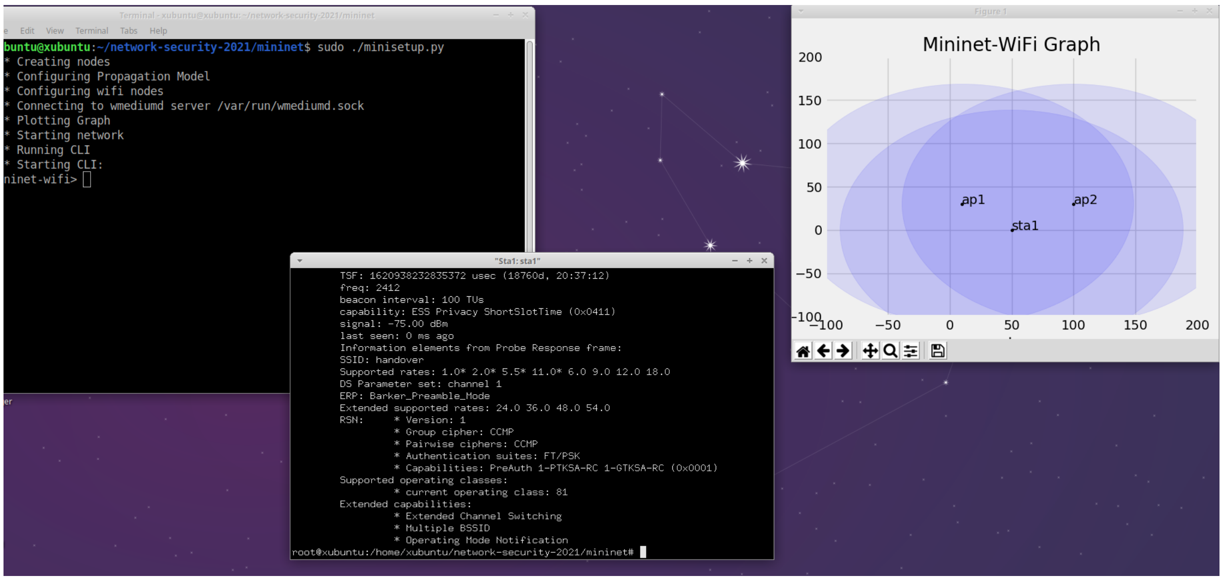
```
xubuntu@xubuntu:~/network-security-2021/mininet$ ./pysetup.sh
Collecting wheel
  Using cached wheel-0.36.2-py2.py3-none-any.whl (35 kB)
Installing collected packages: wheel
Successfully installed wheel-0.36.2
Collecting pycryptodome==3.9.9
  Using cached pycryptodome-3.9.9-cp38-cp38-manylinux1_x86_64.whl (13.7 MB)
Processing /home/xubuntu/.cache/pip/wheels/55/dc/f2/91686c7a482422ab866594a16029
4c457baebab7ad91fd58db/scapy-2.4.4-py2.py3-none-any.whl
Installing collected packages: pycryptodome, scapy
Successfully installed pycryptodome-3.9.9 scapy-2.4.4
xubuntu@xubuntu:~/network-security-2021/mininet$
```

Figure 13: pysetup.sh output

We can now execute the `minisetup.py` file (superuser required for mininet to start):

```
sudo ./minisetup.py
```

This python code will create the virtual mininet-WiFi network also opening the mininet terminal and showing a graphic representation of the network. Finally the script will open a second terminal (xterm) which is dedicated to the station 1 (sta1), at this point we should have the screen look like the next figure.


Terminal - xubuntu@xubuntu: ~/network-security-2021/mininet
e Edit View Terminal Tabs Help
"Sta1: sta1"
Figure 1
Mininet-WiFi Graph
ap1
ap2
sta1
200
150
100
50
0
−50
−100
−100
−50
0
50
100
150
200

```
buntu@xubuntu:~/network-security-2021/mininet$ sudo ./minisetup.py
* Creating nodes
* Configuring Propagation Model
* Configuring wifi nodes
* Connecting to wmediumd server /var/run/wmediumd.sock
* Plotting Graph
* Starting network
* Running CLI
* Starting CLI:
ninet-wifi> 
```

```
        TSF: 1620938232835372 usec (18760d, 20:37:12)
        freq: 2412
        beacon interval: 100 TUs
        capability: ESS Privacy ShortSlotTime (0x0411)
        signal: -75.00 dBm
        last seen: 0 ms ago
        Information elements from Probe Response frame:
        SSID: handover
        Supported rates: 1.0* 2.0* 5.5* 11.0* 6.0 9.0 12.0 18.0
        DS Parameter set: channel 1
        ERP: Barker_Preamble_Mode
        Extended supported rates: 24.0 36.0 48.0 54.0
        RSN:     * Version: 1
                 * Group cipher: CCMP
                 * Pairwise ciphers: CCMP
                 * Authentication suites: FT/PSK
                 * Capabilities: PreAuth 1-PTKSA-RC 1-GTKSA-RC (0x0001)
        Supported operating classes:
                 * current operating class: 81
        Extended capabilities:
                 * Extended Channel Switching
                 * Multiple BSSID
                 * Operating Mode Notification
root@xubuntu:/home/xubuntu/network-security-2021/mininet# 
```

Figure 14: minisetup.py output

As we can see the station 1 terminal already completed a WiFi scan and displays the result. The command used to scan the network is at line 49 of the `minisetup.py` and it is the following:

```
iw dev sta1-wlan0 scan
```

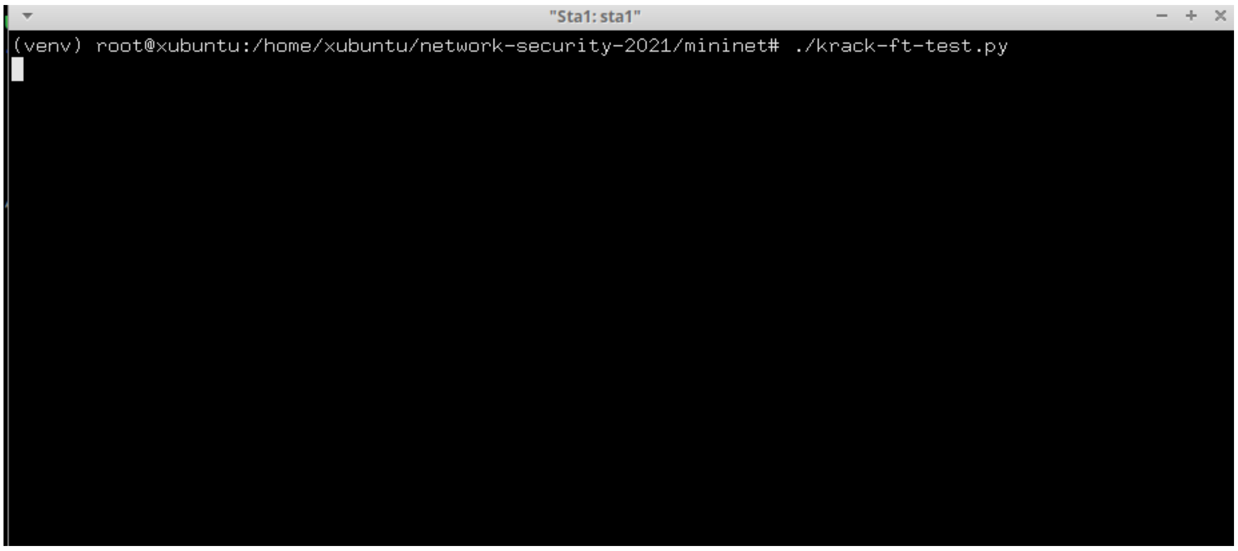
As we can see the station detected both the Access Points, we can confirm this by comparing the returned MAC addresses with the ones assigned in the `minisetup.py` file (line 21 and line 23 respectively for AP1 and AP2).
Returning to our activity, from now on we will work on both the terminals, and can reduce to icon the network graph.
Starting from the station 1 terminal we need to activate the virtual environment, using the following command.

```
source venv/bin/activate
```

When the virtual environment is activated, we can execute the test script

```
./krack-ft-test.py
```

The output is not noticeable yet because the script is waiting for traffic.



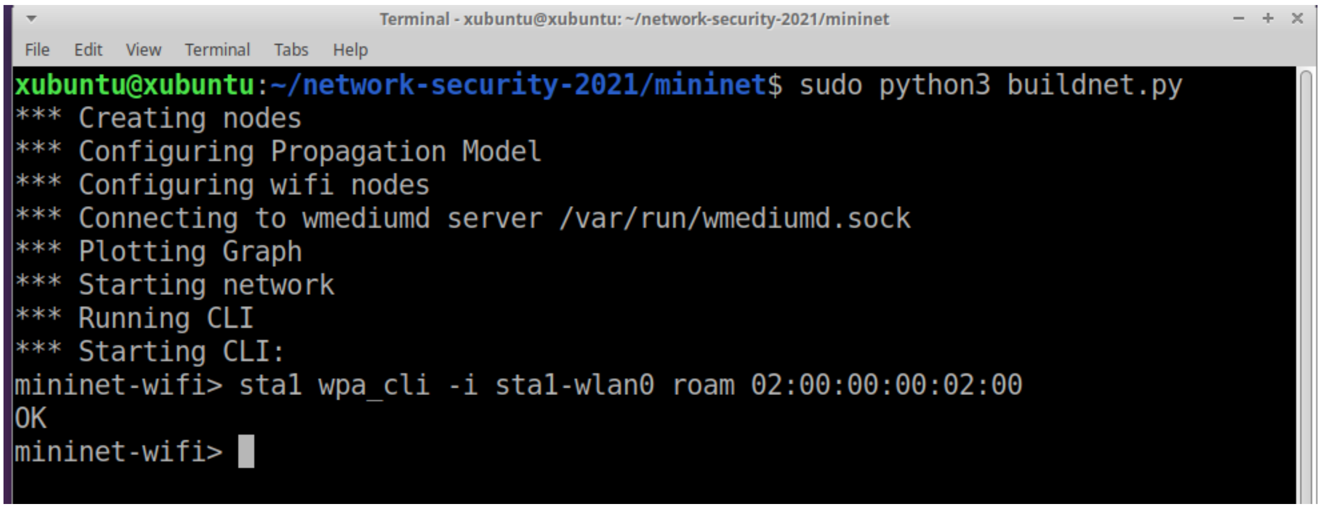
```
(venv) root@xubuntu:/home/xubuntu/network-security-2021/mininet# ./krack-ft-test.py
```

Figure 15: krack-ft-test-py output

Now coming back to the mininet terminal and let's ask to STA1 to roam from AP1 to AP2. It is possible to do that using the following command where we specify the MAC address of AP that the station should reach, in our case AP2.

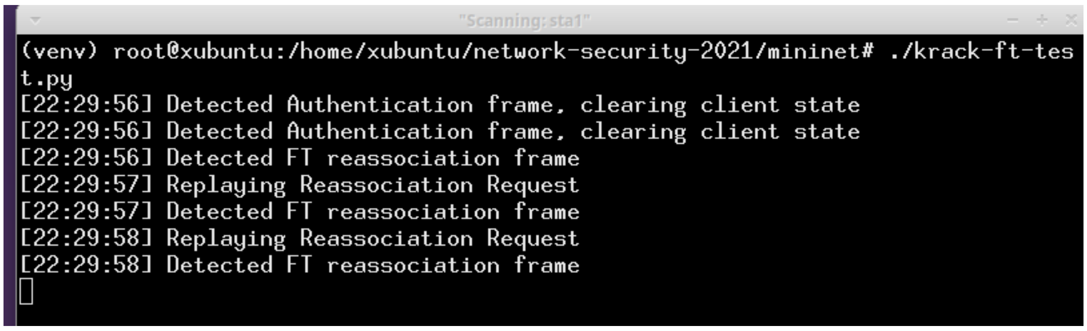
```
sta1 wpa_cli -i sta1_wlan0 roam 02:00:00:00:02:00
```

The mininet terminal should output a simple "OK", whereas in the sta1 terminal the testing script should show that an attempt to replay the reassociation request-response frame is happening. The following two images will show the two outputs.



```
xubuntu@xubuntu:~/network-security-2021/mininet$ sudo python3 buildnet.py
*** Creating nodes
*** Configuring Propagation Model
*** Configuring wifi nodes
*** Connecting to wmediumd server /var/run/wmediumd.sock
*** Plotting Graph
*** Starting network
*** Running CLI
*** Starting CLI:
mininet-wifi> sta1 wpa_cli -i sta1-wlan0 roam 02:00:00:00:02:00
OK
mininet-wifi>
```

Figure 16: Roaming mininet output



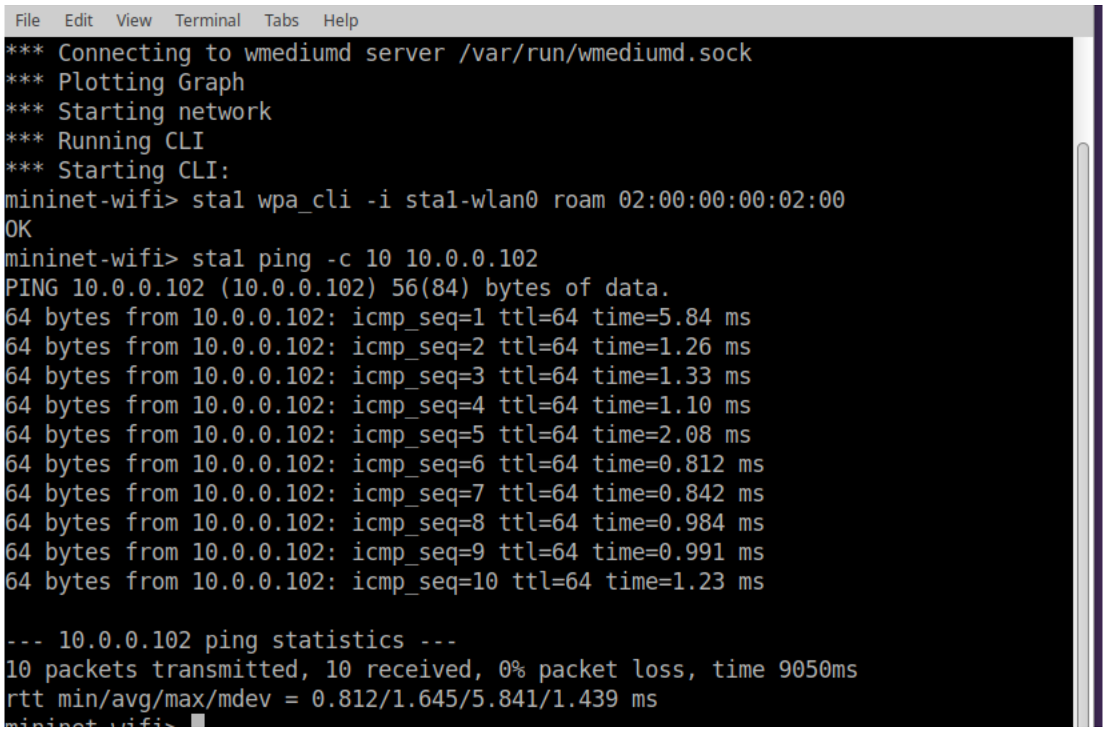
```
(venv) root@xubuntu:/home/xubuntu/network-security-2021/mininet# ./krack-ft-tes
t.py
[22:29:56] Detected Authentication frame, clearing client state
[22:29:56] Detected Authentication frame, clearing client state
[22:29:56] Detected FT reassociation frame
[22:29:57] Replaying Reassociation Request
[22:29:57] Detected FT reassociation frame
[22:29:58] Replaying Reassociation Request
[22:29:58] Detected FT reassociation frame
```

Figure 17: Roaming sta1 output

The final command will let sta1 ping toward the IP address of the second AP. To better show our result we decided to let sta1 ping only ten times, using the -c option of the ping program.

```
sta1 ping -c 10 10.0.0.102
```

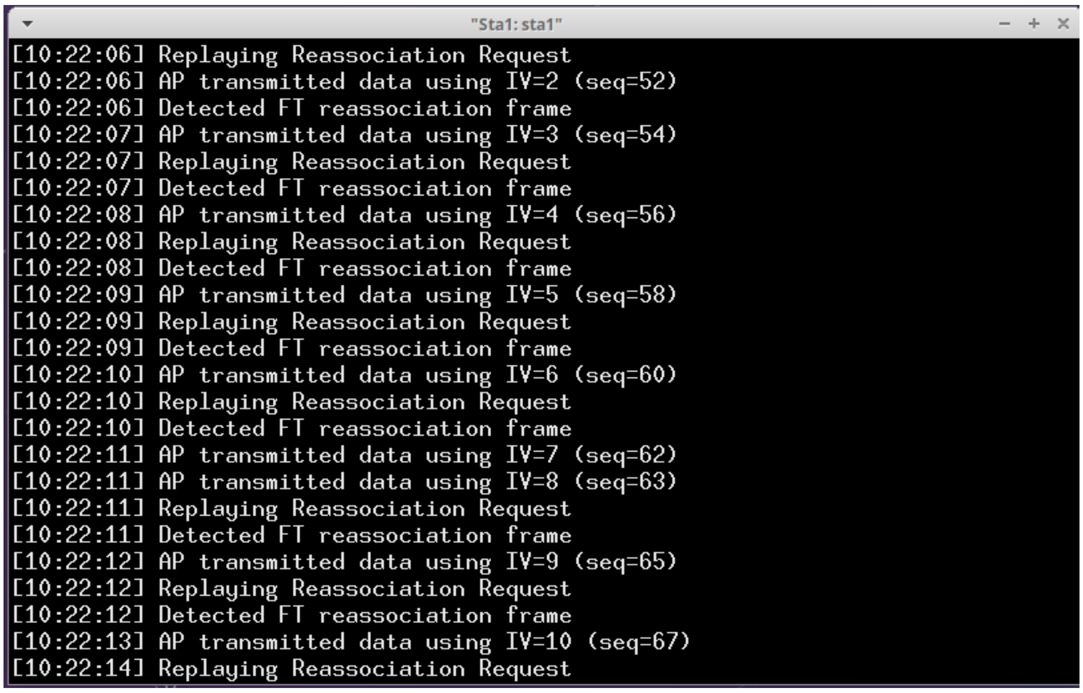
Our result should be the following.



```
*** Connecting to wmediumd server /var/run/wmediumd.sock
*** Plotting Graph
*** Starting network
*** Running CLI
*** Starting CLI:
mininet-wifi> sta1 wpa_cli -i sta1-wlan0 roam 02:00:00:00:02:00
OK
mininet-wifi> sta1 ping -c 10 10.0.0.102
PING 10.0.0.102 (10.0.0.102) 56(84) bytes of data.
64 bytes from 10.0.0.102: icmp_seq=1 ttl=64 time=5.84 ms
64 bytes from 10.0.0.102: icmp_seq=2 ttl=64 time=1.26 ms
64 bytes from 10.0.0.102: icmp_seq=3 ttl=64 time=1.33 ms
64 bytes from 10.0.0.102: icmp_seq=4 ttl=64 time=1.10 ms
64 bytes from 10.0.0.102: icmp_seq=5 ttl=64 time=2.08 ms
64 bytes from 10.0.0.102: icmp_seq=6 ttl=64 time=0.812 ms
64 bytes from 10.0.0.102: icmp_seq=7 ttl=64 time=0.842 ms
64 bytes from 10.0.0.102: icmp_seq=8 ttl=64 time=0.984 ms
64 bytes from 10.0.0.102: icmp_seq=9 ttl=64 time=0.991 ms
64 bytes from 10.0.0.102: icmp_seq=10 ttl=64 time=1.23 ms

--- 10.0.0.102 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9050ms
rtt min/avg/max/mdev = 0.812/1.645/5.841/1.439 ms
```

Figure 18: Ping sta1 output

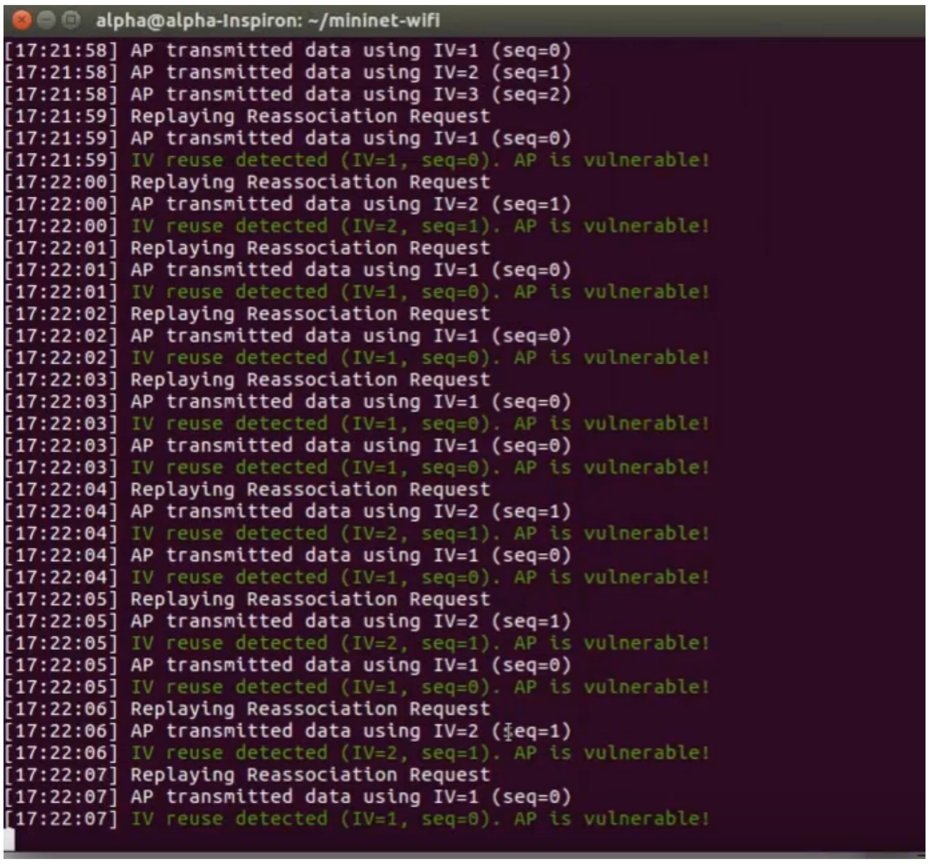
At the end of the ping process we can analyze the result in the station 1 terminal. As we can see the attempt of re-association was not successful and the Initialization Vector is never reused.



```
[10:22:06] Replaying Reassociation Request
[10:22:06] AP transmitted data using IV=2 (seq=52)
[10:22:06] Detected FT reassociation frame
[10:22:07] AP transmitted data using IV=3 (seq=54)
[10:22:07] Replaying Reassociation Request
[10:22:07] Detected FT reassociation frame
[10:22:08] AP transmitted data using IV=4 (seq=56)
[10:22:08] Replaying Reassociation Request
[10:22:08] Detected FT reassociation frame
[10:22:09] AP transmitted data using IV=5 (seq=58)
[10:22:09] Replaying Reassociation Request
[10:22:09] Detected FT reassociation frame
[10:22:10] AP transmitted data using IV=6 (seq=60)
[10:22:10] Replaying Reassociation Request
[10:22:10] Detected FT reassociation frame
[10:22:11] AP transmitted data using IV=7 (seq=62)
[10:22:11] AP transmitted data using IV=8 (seq=63)
[10:22:11] Replaying Reassociation Request
[10:22:11] Detected FT reassociation frame
[10:22:12] AP transmitted data using IV=9 (seq=65)
[10:22:12] Replaying Reassociation Request
[10:22:12] Detected FT reassociation frame
[10:22:13] AP transmitted data using IV=10 (seq=67)
[10:22:14] Replaying Reassociation Request
```

Figure 19: sta1 output

To better understand our result, we can compare our output with the output of the same test made in a vulnerable environment. As we can see from the next screen is clear how after the re-association request-response frames the Initialization Vector was reused.



```
[17:21:58] AP transmitted data using IV=1 (seq=0)
[17:21:58] AP transmitted data using IV=2 (seq=1)
[17:21:58] AP transmitted data using IV=3 (seq=2)
[17:21:59] Replaying Reassociation Request
[17:21:59] AP transmitted data using IV=1 (seq=0)
[17:21:59] IV reuse detected (IV=1, seq=0). AP is vulnerable!
[17:22:00] Replaying Reassociation Request
[17:22:00] AP transmitted data using IV=2 (seq=1)
[17:22:00] IV reuse detected (IV=2, seq=1). AP is vulnerable!
[17:22:01] Replaying Reassociation Request
[17:22:01] AP transmitted data using IV=1 (seq=0)
[17:22:01] IV reuse detected (IV=1, seq=0). AP is vulnerable!
[17:22:02] Replaying Reassociation Request
[17:22:02] AP transmitted data using IV=1 (seq=0)
[17:22:02] IV reuse detected (IV=1, seq=0). AP is vulnerable!
[17:22:03] Replaying Reassociation Request
[17:22:03] AP transmitted data using IV=1 (seq=0)
[17:22:03] IV reuse detected (IV=1, seq=0). AP is vulnerable!
[17:22:03] AP transmitted data using IV=1 (seq=0)
[17:22:03] IV reuse detected (IV=1, seq=0). AP is vulnerable!
[17:22:04] Replaying Reassociation Request
[17:22:04] AP transmitted data using IV=2 (seq=1)
[17:22:04] IV reuse detected (IV=2, seq=1). AP is vulnerable!
[17:22:04] AP transmitted data using IV=1 (seq=0)
[17:22:04] IV reuse detected (IV=1, seq=0). AP is vulnerable!
[17:22:05] Replaying Reassociation Request
[17:22:05] AP transmitted data using IV=2 (seq=1)
[17:22:05] IV reuse detected (IV=2, seq=1). AP is vulnerable!
[17:22:05] AP transmitted data using IV=1 (seq=0)
[17:22:05] IV reuse detected (IV=1, seq=0). AP is vulnerable!
[17:22:06] Replaying Reassociation Request
[17:22:06] AP transmitted data using IV=2 (seq=1)
[17:22:06] IV reuse detected (IV=2, seq=1). AP is vulnerable!
[17:22:07] Replaying Reassociation Request
[17:22:07] AP transmitted data using IV=1 (seq=0)
[17:22:07] IV reuse detected (IV=1, seq=0). AP is vulnerable!
```

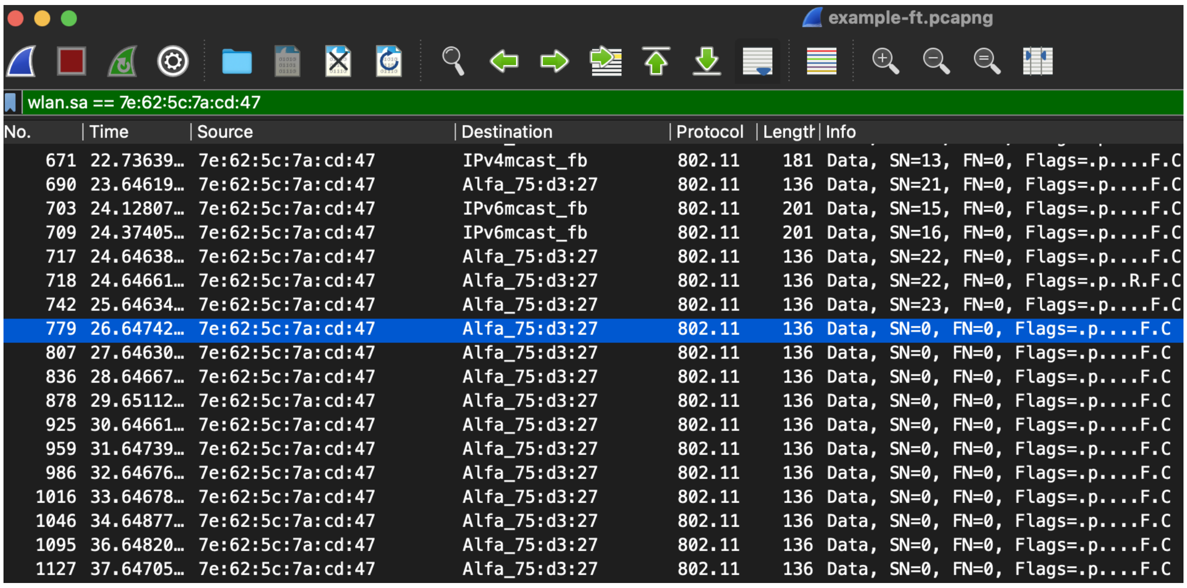
Figure 20: example of vulnerable output

2.3.2 Simulation on a vulnerable system

Unfortunately we were not able to setup a vulnerable network in mininet-WiFi, this is due to the fact that in order to have a vulnerable AP we need to mix together and compile a new version on mininet which supports all the python commands required to effectively build the network and a vulnerable (old) version of hostapd, everything needs to run on an unpatched linux kernel, for example Ubuntu 17.10 initial release or 16.04. The problem of meeting all those requirements is that in some cases mininet will not work completely and in other cases the test will still show a patched system, we tried to solve this for a few days by communicating also with one of the mininet-WiFi main developers (Ramon Fontes) but we could not find a solution.

2.4 Packet captures

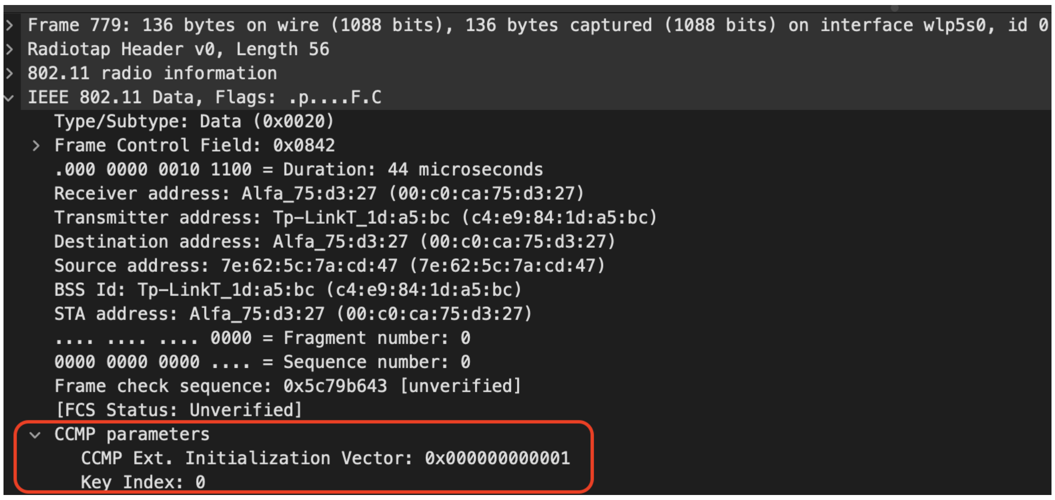
We will now analyze a pre-recorded packet capture inside wireshark to spot the attack. The `example-ft.pcapng` file is contained into the captures folder, we can simply open it with wireshark and then filter out the useless packet by applying the filter shown in the next figure.



```
wlan.sa == 7e:62:5c:7a:cd:47
No.   Time        Source               Destination    Protocol Length Info
 671  22.73639…  7e:62:5c:7a:cd:47    IPv4mcast_fb   802.11   181  Data, SN=13, FN=0, Flags=.p....F.C
 690  23.64619…  7e:62:5c:7a:cd:47    Alfa_75:d3:27  802.11   136  Data, SN=21, FN=0, Flags=.p....F.C
 703  24.12807…  7e:62:5c:7a:cd:47    IPv6mcast_fb   802.11   201  Data, SN=15, FN=0, Flags=.p....F.C
 709  24.37405…  7e:62:5c:7a:cd:47    IPv6mcast_fb   802.11   201  Data, SN=16, FN=0, Flags=.p....F.C
 717  24.64638…  7e:62:5c:7a:cd:47    Alfa_75:d3:27  802.11   136  Data, SN=22, FN=0, Flags=.p....F.C
 718  24.64661…  7e:62:5c:7a:cd:47    Alfa_75:d3:27  802.11   136  Data, SN=22, FN=0, Flags=.p..R.F.C
 742  25.64634…  7e:62:5c:7a:cd:47    Alfa_75:d3:27  802.11   136  Data, SN=23, FN=0, Flags=.p....F.C
 779  26.64742…  7e:62:5c:7a:cd:47    Alfa_75:d3:27  802.11   136  Data, SN=0, FN=0, Flags=.p....F.C
 807  27.64630…  7e:62:5c:7a:cd:47    Alfa_75:d3:27  802.11   136  Data, SN=0, FN=0, Flags=.p....F.C
 836  28.64667…  7e:62:5c:7a:cd:47    Alfa_75:d3:27  802.11   136  Data, SN=0, FN=0, Flags=.p....F.C
 878  29.65112…  7e:62:5c:7a:cd:47    Alfa_75:d3:27  802.11   136  Data, SN=0, FN=0, Flags=.p....F.C
 925  30.64661…  7e:62:5c:7a:cd:47    Alfa_75:d3:27  802.11   136  Data, SN=0, FN=0, Flags=.p....F.C
 959  31.64739…  7e:62:5c:7a:cd:47    Alfa_75:d3:27  802.11   136  Data, SN=0, FN=0, Flags=.p....F.C
 986  32.64676…  7e:62:5c:7a:cd:47    Alfa_75:d3:27  802.11   136  Data, SN=0, FN=0, Flags=.p....F.C
1016  33.64678…  7e:62:5c:7a:cd:47    Alfa_75:d3:27  802.11   136  Data, SN=0, FN=0, Flags=.p....F.C
1046  34.64877…  7e:62:5c:7a:cd:47    Alfa_75:d3:27  802.11   136  Data, SN=0, FN=0, Flags=.p....F.C
1095  36.64820…  7e:62:5c:7a:cd:47    Alfa_75:d3:27  802.11   136  Data, SN=0, FN=0, Flags=.p....F.C
1127  37.64705…  7e:62:5c:7a:cd:47    Alfa_75:d3:27  802.11   136  Data, SN=0, FN=0, Flags=.p....F.C
```

Figure 21: Wireshark captures

By looking at the packets from 779 to 1127 we can notice that all use the CCMP IV value 1, this was caused by malicious retransmissions of the FT reassociation request.



```
> Frame 779: 136 bytes on wire (1088 bits), 136 bytes captured (1088 bits) on interface wlp5s0, id 0
> Radiotap Header v0, Length 56
> 802.11 radio information
v IEEE 802.11 Data, Flags: .p....F.C
    Type/Subtype: Data (0x0020)
  > Frame Control Field: 0x0842
    .000 0000 0010 1100 = Duration: 44 microseconds
    Receiver address: Alfa_75:d3:27 (00:c0:ca:75:d3:27)
    Transmitter address: Tp-LinkT_1d:a5:bc (c4:e9:84:1d:a5:bc)
    Destination address: Alfa_75:d3:27 (00:c0:ca:75:d3:27)
    Source address: 7e:62:5c:7a:cd:47 (7e:62:5c:7a:cd:47)
    BSS Id: Tp-LinkT_1d:a5:bc (c4:e9:84:1d:a5:bc)
    STA address: Alfa_75:d3:27 (00:c0:ca:75:d3:27)
    .... .... .... 0000 = Fragment number: 0
    0000 0000 0000 .... = Sequence number: 0
    Frame check sequence: 0x5c79b643 [unverified]
    [FCS Status: Unverified]
  v CCMP parameters
      CCMP Ext. Initialization Vector: 0x000000000001
      Key Index: 0
```

Figure 22: CCMP IV