Connected Car Hacking

In-Vehicle Infotainment (IVI) unit hacking

hardwear.io

Quarkslab

Overview

- **In-Vehicle Infotainment (IVI)** unit is the ECU in charge of most of the multimedia and customer-oriented features inside the vehicle

- Primary purpose is to enhance the driving experience by offering a **wide range of features**:
  - Navigation
  - Entertainment
  - Connected applications
  - Interaction with driver/passenger smartphones

- This ECU has the widest range of **attack surfaces**, being a target of choice

Illustration: [link]

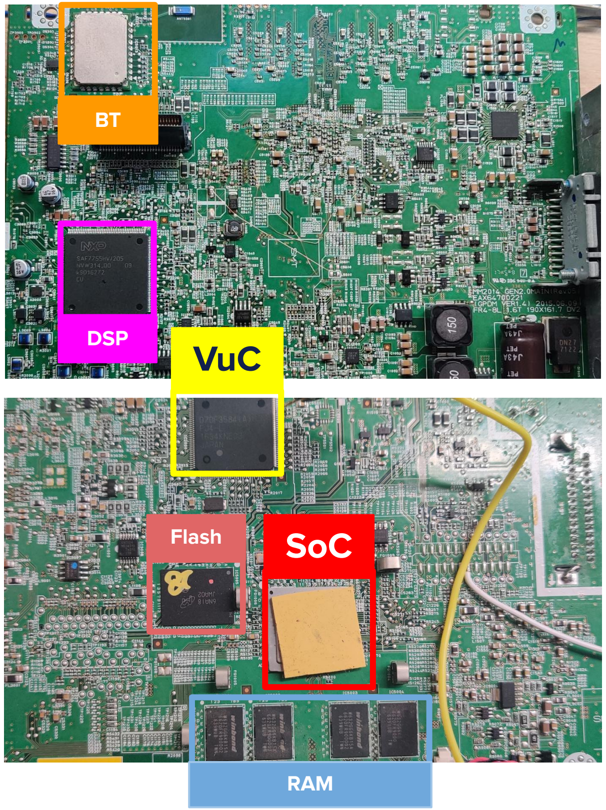
Hardware analysis

Key elements of an **IVI** are:

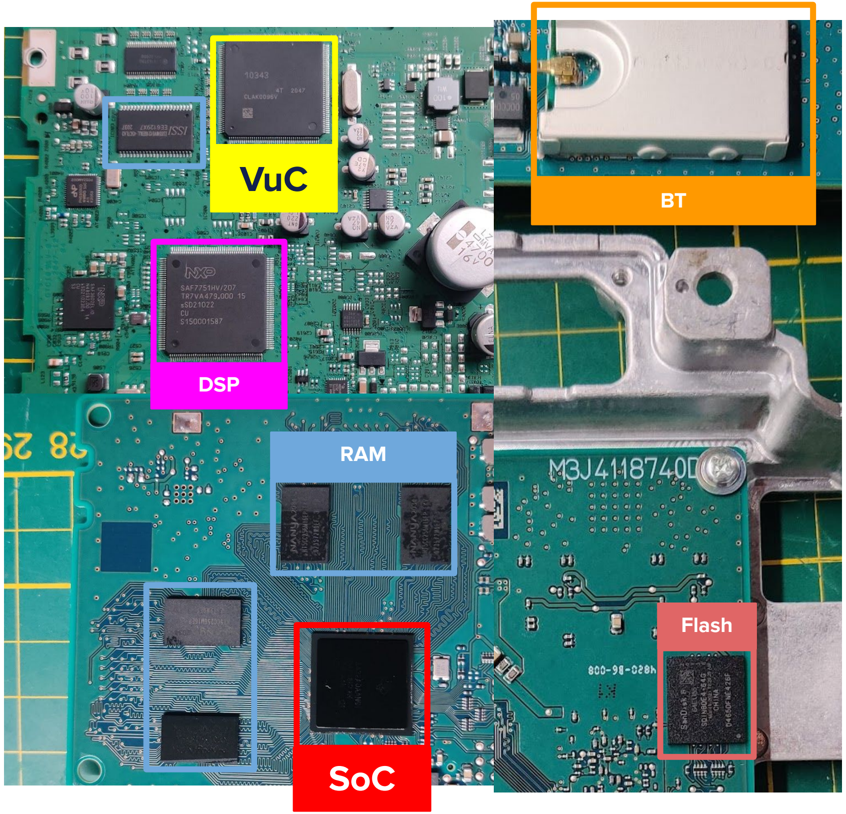
- **Dedicated vehicle microcontroller:** or **VuC** (Vehicle uC), which interacts with the on-board networks like CAN and handles diagnostic requests
- **Main System on Chip (SoC):** runs the operating system (OS)
- **External Flash memory and RAM:** for the SoC
- **Digital Signal Processor (DSP):** process audio signals
- **External connectivity chip**: for wireless connectivity and positioning (GNSS, Wi-Fi, Bluetooth, …)
- Some IVI have **integrated display**, others are separated in 2 ECUs, connected with **LVDS cable (Low Voltage Differential Signal)**

Hardware analysis

IVI #A



IVI #B



IVI Firmware extraction

Quarkslab

Embedded firmwares

- To fully assess an IVI at least 2 firmwares need to be analysed:
  - **VuC** with its **embedded firmware** for in-vehicle communications
  - **SoC** running an **operating system**, which manages wireless/USB connectivity and system updates

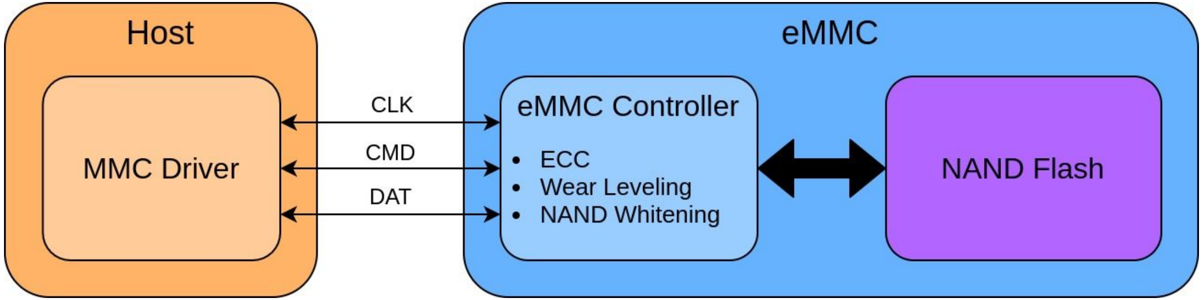
- Main operating system used in IVI are:
  - **Android**
  - **QNX**
  - **Automotive Grade Linux**

- The operating system is commonly store in on an **eMMC** memory chip

Firmware storage - eMMC

To store an Operating System, **eMMC** chip are often used.

eMMC have a dedicated controller that handles the inner **NAND Flash** memory and performs wear levelling and error correction code of this memory.



The various pin of an **eMMC** are:

- **CMD** - command line
- **CLK** - clock line
- **DAT[0-7]** - data lines
- **VCC** - input voltage for the flash storage (1.8V and/or 3.3V)
- **VCCQ** - input voltage for the flash controller (1.8V and/or 3.3V)

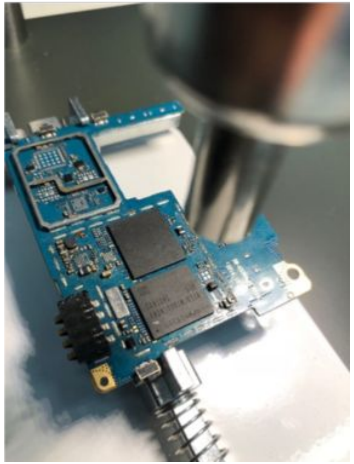
Illustration: [link]

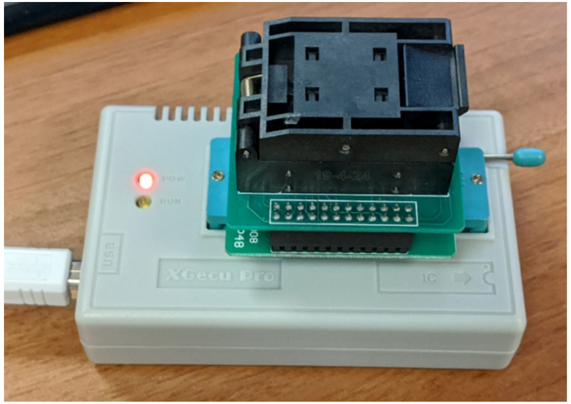
Firmware storage - eMMC extraction

eMMC use the same protocol as **SD-card**, and could operate in 1-bit, 4-bit or 8-bit mode.

Content of an **eMMC** could be extracted by removing the chip from the PCB using **chip-off** technique and a memory programmer.

However, by finding pins **CMD**, **CLK** and **DAT[0-4]**, it is possible to solder them onto an SD adapter and read and write it using an SD-card reader. This technique is useful to keep the device working and inject/extract data and binaries. To avoid conflict on the **CLK** line, the main chip of the device need to be **halted** or **unpowered**.





Secure Digital Input Output

Secure Digital Input Output, or **SDIO**, is synchronous protocol enabling high-speed data transfer with peripheral devices.

It requires a **command** (CMD), a **clock** (CLK) line and **several data lines** (DATx) to supports **8-bit, 4-bit or 1-bit data bus**.

It provides faster speeds than SPI and I2C, making it ideal for high bandwidth applications.

This protocol is used by SD card and eMMC memory chip.

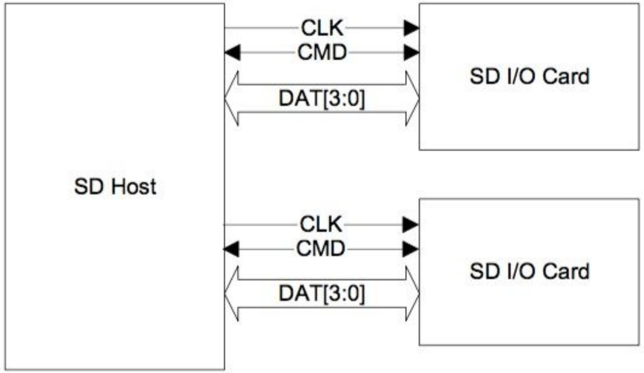


Figure 2-1 Signal connection to two 4-bit SDIO cards

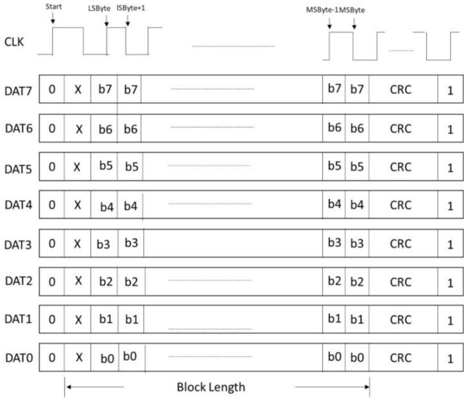
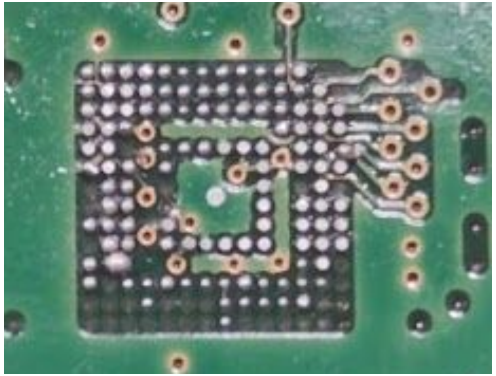


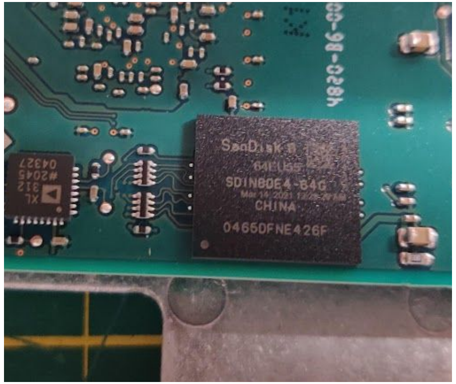
Illustration: [link] & [link]

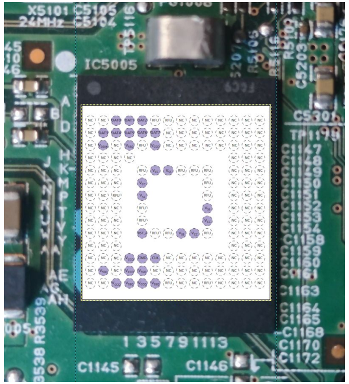
Firmware storage - eMMC pins identification

To find the required pins to perform the extraction using a SD-card reader, there are several techniques:

- **Removing the eMMC** on a second device, then find vias, debug pins or components to solder to
- Search for **array of pull-up/pull-down resistors** around the eMMC, which are good candidates for **DAT** pins
- **Align pictures** of the two sides of PCB then **overlay the pins** of a spare eMMC over the real one to visually found candidate traces



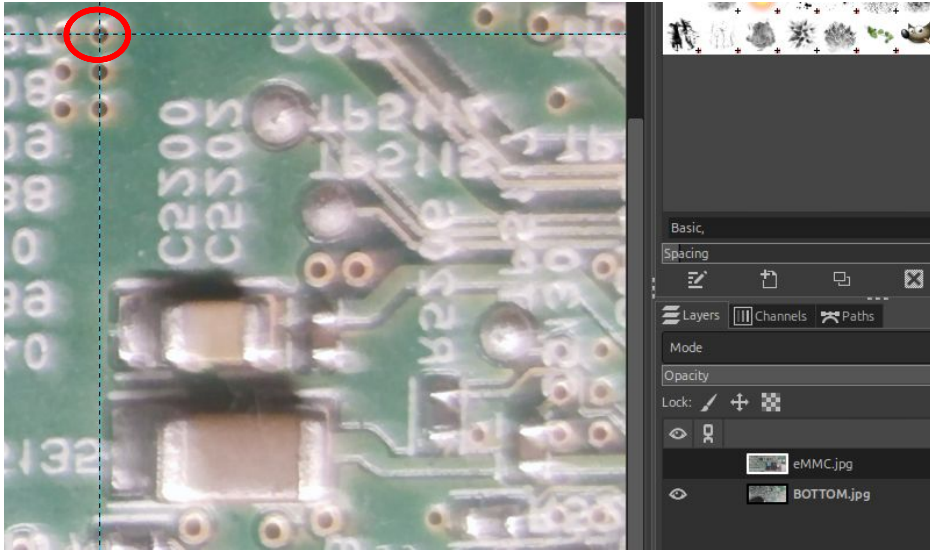


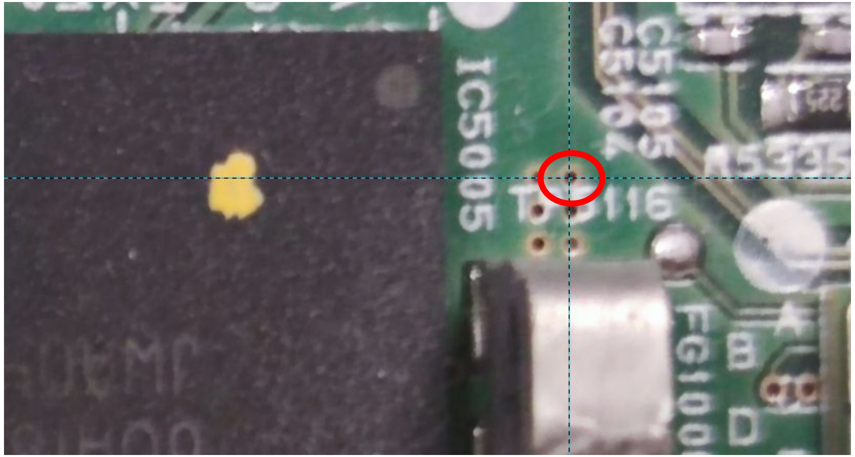


Firmware storage - eMMC pins identification

To overlay and align images, we will use **GIMP** and its **unified transform tool**

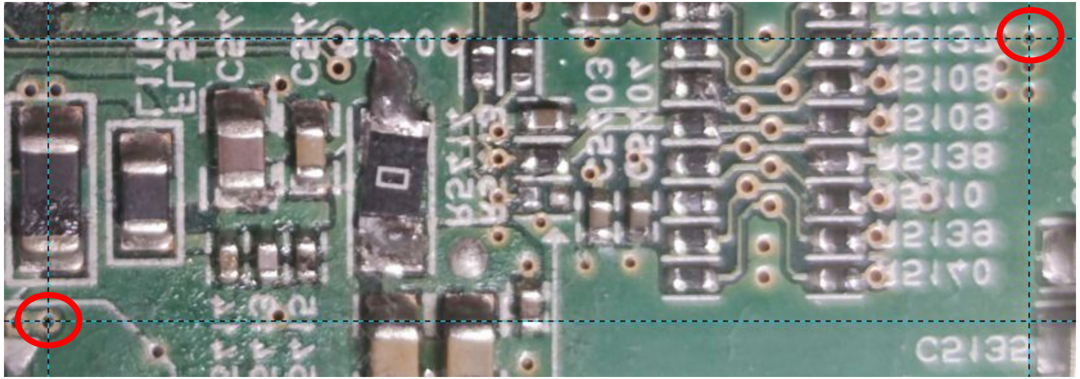
- **1.** Open one of the picture of the PCB and add the second one in a new layer
- **2**. Using **guides**, **mark a 1st reference** (an easily identifiable vias for example) on the bottom layer
- **3**. **Align the upper layer** on this first mark

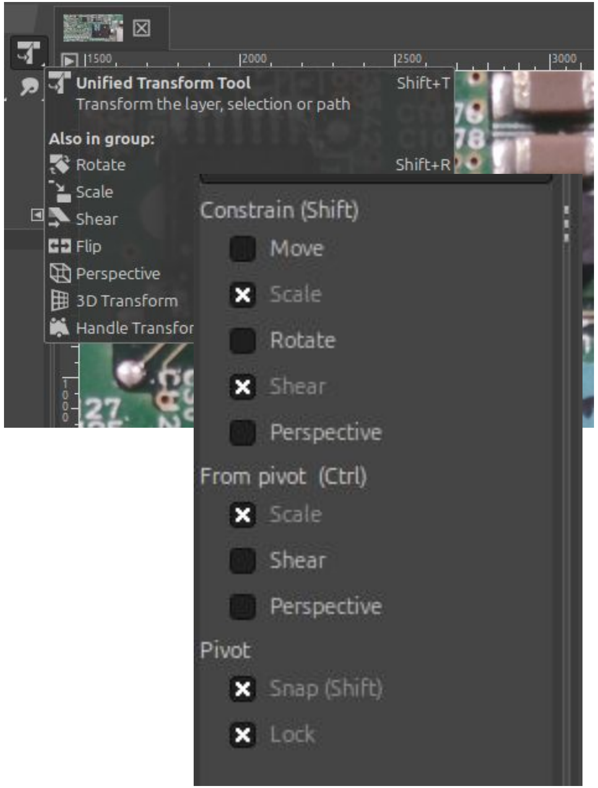




Firmware storage - eMMC pins identification

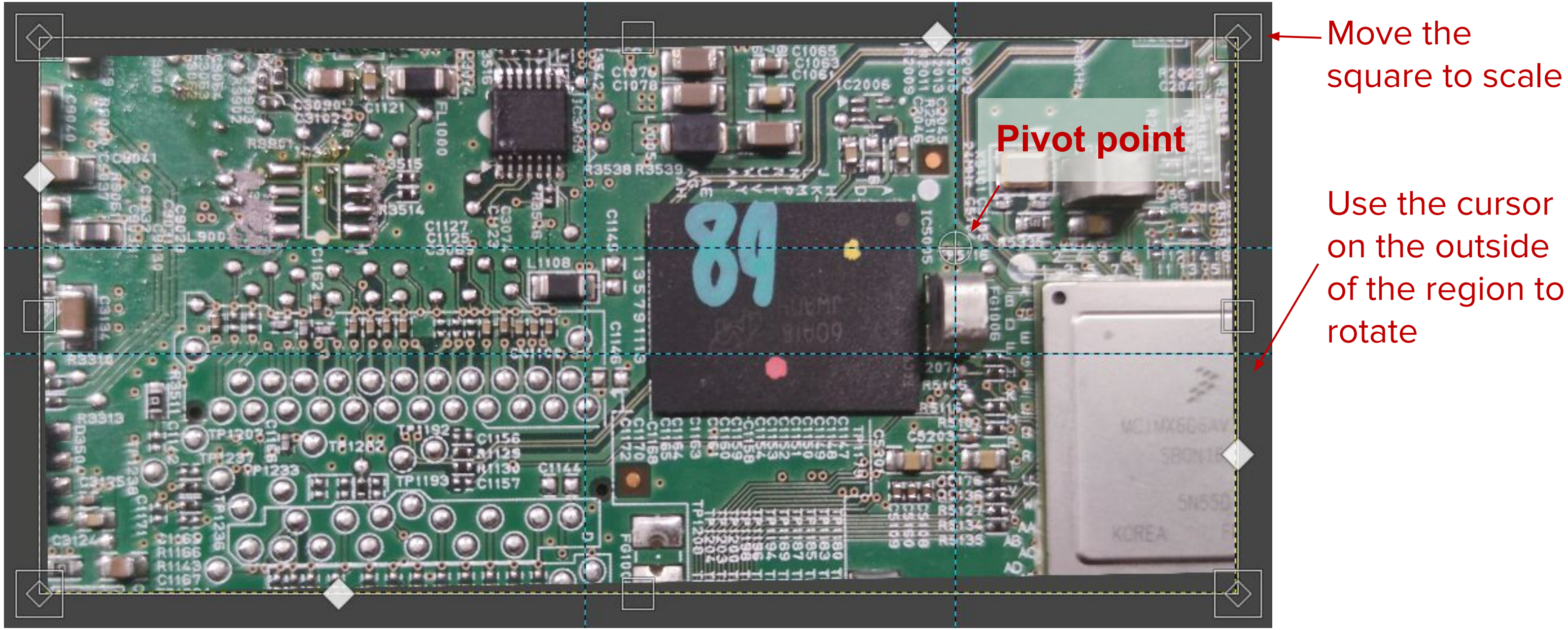
- **4.** On the bottom image, **select a second reference point**
- **5**. Open the **Unified Transform Tool**
- **6**. Check **Pivot > Lock**, **Pivot > Snap** and **From Pivot > Scale**





Firmware storage - eMMC pins identification

- **7.** Click on the upper image to activate the **Unified Transform Tool**
- **8**. Move the **pivot point** (crosshair) on the **first reference point**
- **6. Scale** and **rotate** the image to match the **second reference point**



Lab 1 - Locating eMMC pins

Goals

- For this lab, you'll practice visual identification of the **PCB** of an **IVI** to find components or test points that could be used to extract the **eMMC** memory
- Complete challenges **eMMC identification** to **eMMC pin identification 2** in **Infotainement - firmware extraction**

Firmware storage - eMMC pins identification - initialisation

- It is also possible to find candidates using a **logic analyser**
- In 1-byte mode, only **CMD**, **CLK** and **DAT0** are required
- During **initialisation**, DAT0 is the only data line active
- Commands sent through the **CMD** line use the following structure:

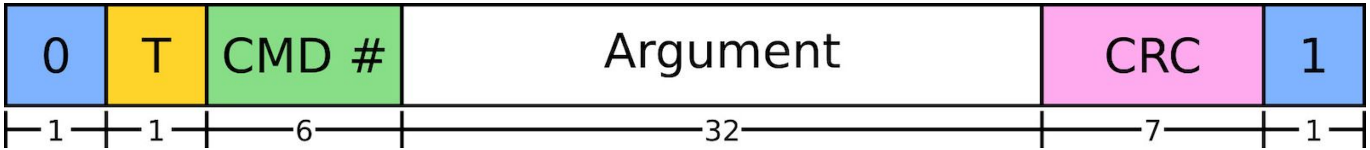


Illustration: [link]

Lab 2 - Identifying eMMC CMD/CLK/DAT0

Goals

- A **logic analyser capture** was performed on test points around an eMMC during boot
- Complete challenge **Signal analysis** from **Infotainment - firmware extraction**

Firmware storage - eMMC extraction - disabling auto-mount

- Using a **SD reader**, it is possible to mount the eMMC on your computer
- Beware of the **voltage** of the eMMC, check the datasheet, some only work on **1.8V**, requiring special adapter
- Also, it is recommended to **disable auto-mount**, to avoid writing metadata on the eMMC, which can trigger secure boot protection
- On Debian, type:

```
$ sudo systemctl stop udisks2
$ sudo systemctl disable udisks2
```

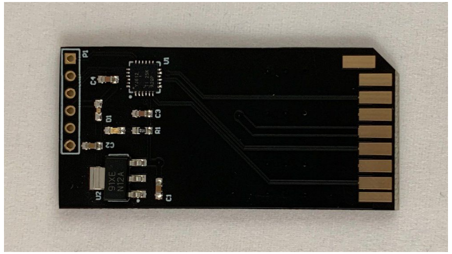


Illustration: [link]

Firmware storage - eMMC extraction - dump

- Use **dmesg** command to find the path to the eMMC (**/dev/sdX**, **/dev/mmcblkX**)
- The last number is the **partition**, omitting it will dump the whole content of the eMMC

```
$ sudo dmesg
[18989.012121] scsi 0:0:0:0: Direct-Access     Multiple Card  Reader     1.00 PQ: 0 ANSI: 0
[18989.012421] sd 0:0:0:0: Attached scsi generic sg0 type 0
[18989.232914] sd 0:0:0:0: [sda] 15204352 512-byte logical blocks: (7.78 GB/7.25 GiB)
[18989.233345] sd 0:0:0:0: [sda] Write Protect is off
[18989.233348] sd 0:0:0:0: [sda] Mode Sense: 03 00 00 00
[18989.233728] sd 0:0:0:0: [sda] No Caching mode page found
[18989.233733] sd 0:0:0:0: [sda] Assuming drive cache: write through
[18989.236785]  sda: sda1 sda2 sda3
[18989.237038] sd 0:0:0:0: [sda] Attached SCSI removable disk
[18989.525477]  sda: sda1 sda2 sda3          <- 3 partitions


$ sudo dd if=/dev/sda of=./dump.bin bs=512 status=progress     <- Full dump
```

Firmware storage - eMMC extraction - partial

- Using **fdisk**, you can list the various partitions to select a target one to speed up the dumping process

```
$ sudo fdisk -l
Device      Boot   Start        End         Sectors     Size        Id Type
/dev/sdb1                8192      532479      524288    256M   c W95 FAT32 (LBA)
/dev/sdb2              532480     6846463     6313984      3G  83 Linux
/dev/sdb3             6846464     7434239      587776    287M   83 Linux
/dev/sdb4             7434240    31115263    23681024   11,3G   83 Linux

$ sudo dd if=/dev/sda of=./dump.bin bs=512 status=progress seek=6846464 count=587776
```

Lab 3 - Dumping eMMC

Goals

- Using the provided **SD card adapter**, dump the content of the **eMMC**
- Complete remaining **Infotainment - firmware extraction** challenges

Bypassing secure boot

Quarkslab

Infotainment main OS

- Infotainment unit commonly runs **Linux based distribution**, like:
  - **QNX**
  - **Automotive Grade Linux**
  - **Android Automotive OS**

- For this module, we will focus on **Android Automotive** as it has some specifics

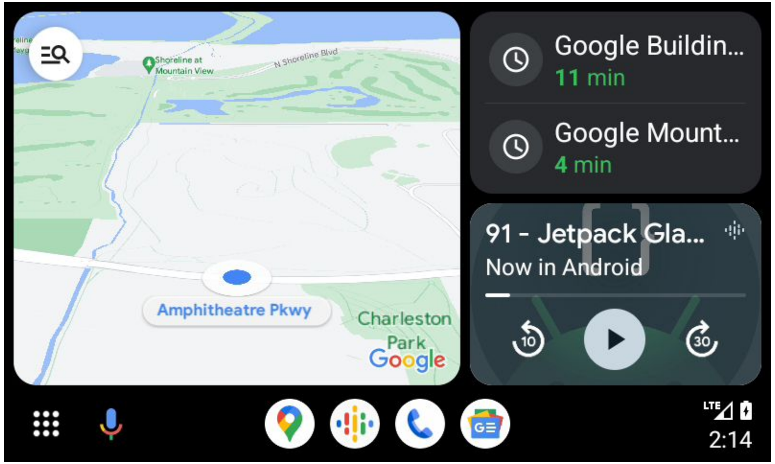


Illustration: [link]

Firmware alteration

- Having read/write access to the eMMC, we can modify its content to add/patch **binaries** or **modify configuration files**
- Correctly secured IVI will implement a **secure boot** to ensure the authenticity of the OS
- But … it has to be done in a proper way
- Not all partition could be **signed**, as at least one partition must be alterable to **store persistent/user data**
- What if this partition is **executable** or store **binaries/configuration** used by the system ?
- Analysing **logs** may give some hints on how the **secure boot** is implemented

Standard secure boot process

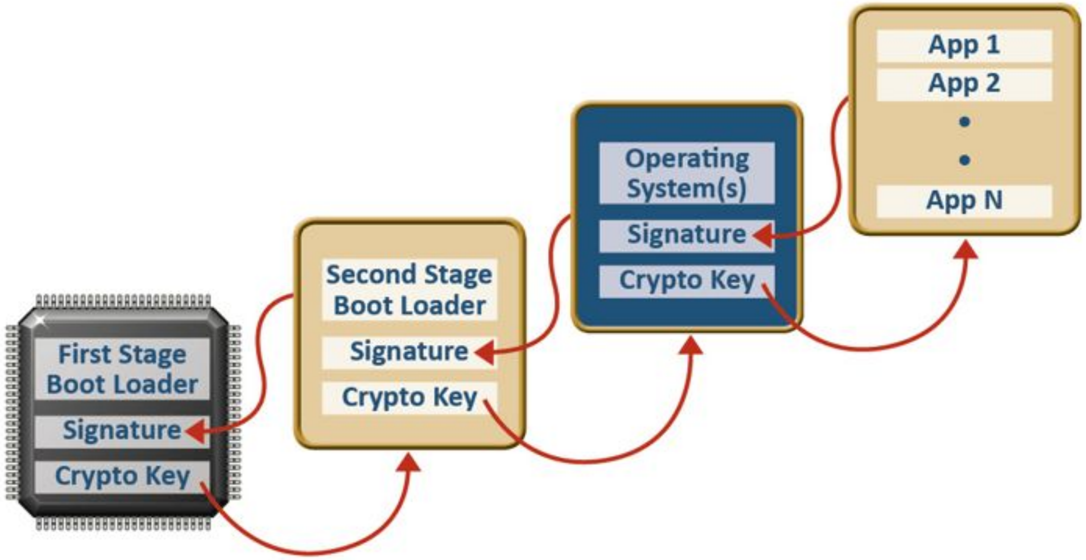


Illustration: [link]

Lab 4 - Bypassing vulnerable secure boot

Goals

- Can you bypass the secure boot of a 10 years old infotainment ?
- Complete challenges **Infotainment - bypassing secure boot**

Android 101

Quarkslab

Android Automotive specifics

- In **Android Automotive**, applications are **car-optimized** for safety in road usafe
- It handles **interaction** with some part of the vehicle, including:
  - HVAC
  - Instrument cluster display (navigation, music)
  - Battery charging
  - Locking system
  - Telematic Control Unit
- It can support some base Android services, known as **Google Android Automotive Services (GAS)**, like the Play Store, Maps…
- If it offers different features, it is based on the **Android Open Source Project** (AOSP)

Android Automotive - Per-Application Network Selection

- **Per-Application Network Selection** is an API that is specific to **Android Automotive**
- It allows network management to route data traffic of **OEM-allowed application** to **OEM network**
- It ensure **critical applications** could be connected using an **OEM-paid data plan** and other application to use **user-paid data plan**
- It expands **NetworkCapabilities** by adding **NET_CAPABILITY_OEM_PAID** and **NET_CAPABILITY_OEM_PRIVATE**

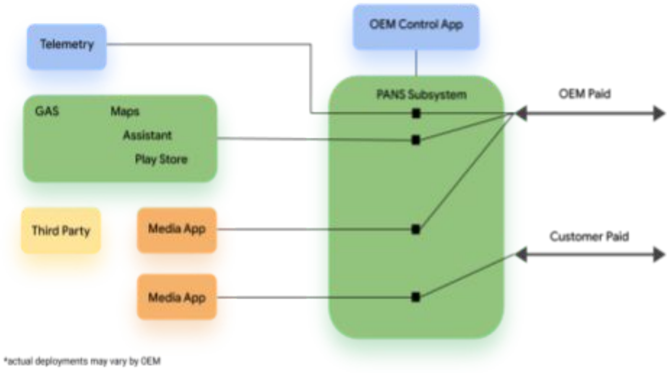


Illustration: [link]

Android key concepts

- **Android** runs applications in a **sandboxed environment** (per-application UID)
- It has **permission-based** access control
- Applications can communicate through **Inter-Process Communication (IPC)**
- Core packages, applications and services are signed using **keys**, granting specific privileges
- The key that signed core packages is known as the **platform key**
- **ADB** is a tool allowing shell access to an Android device, that is normally disabled in production, also known as **user mode**

APK 101

- In **Android**, Applications and Services are provided in as APK, which are ZIP archive
- Key element of an APK is its **AndroidManifest.xml**, which defines application behavior, components, permissions…
- **JADX** allows to unpack **APK** and analysis **Java/Kotlin** code
- An APK could also integrates **native library** which are C/C++ compiled code developed for specific uses
- By default, **applications are restricted**, unless proper permissions are set (Internet access, read/write access to shared folder…)
- Basic APK have no privilege, **unless signed by the platform key,** being then considered as system application

Android Manifest

- The **Android Manifest** is an XML file describing the behavior of an APK and its permissions
- Some **permissions** could be dangerous from a security point of view
- It also declares the **BroadcastReceivers**, for **Intents** that could be use to interact with the application

Android Manifest - example

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"  package="com.example.app">
    <!-- Permissions -->
    <uses-permission android:name="android.permission.INTERNET" />
    <application
        android:allowBackup="true"
        android:label="ExampleApp"
        android:usesCleartextTraffic="true"
        android:theme="@style/Theme.App">
        <!-- Receiver for system BOOT event -->
        <receiver android:name=".BootReceiver"
                  android:enabled="true"
                  android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.BOOT_COMPLETED" />
            </intent-filter>
        </receiver>
        <!-- Receiver for a custom Intent -->
        <receiver android:name=".CustomReceiver"
                  android:enabled="true"
                  android:exported="true">
            <intent-filter>
                <action android:name="com.example.CUSTOM_ACTION" />
            </intent-filter>
        </receiver>
    </application>
</manifest>
```

Android’s quick win: Intents

- For **Inter-Process Communication** application could use **Intents**
- Incorrectly secured **Intents** could allows non-privileged application to execute command or alter the behavior of the target application
- Exposed **BroadcastReceivers** can be found by looking at the various **AndroidManifest.xml**

Sending Intent:

```
Intent sendIntent = new Intent();
sendIntent.setAction(Intent.ACTION_SEND);
sendIntent.putExtra(Intent.EXTRA_TEXT, textMessage);
sendIntent.setType("text/plain");
startActivity(sendIntent);
```

Receiving Intent:

```
MyBroadcastReceiver myBroadcastReceiver = new MyBroadcastReceiver();
IntentFilter filter = new IntentFilter("com.example.snippets.ACTION_UPDATE_DATA");
ContextCompat.registerReceiver(context, myBroadcastReceiver, filter, receiverFlags);
```

Android's quick win: Services & AIDL

- Also **Inter-Process Communication**, **Services** can have **AIDL interfaces**
- They will expose a set of functions that can be executed through **ADB** access or by **third-party APK**
- As the **AIDL** file will not be available in the compiled application, look for and switch-case

AIDL example:

```
    // IRemoteService.aidl
    package com.example.android;


    /** Example service interface */
    interface IRemoteService {
        /** Request the process ID of this service. */
        int getPid();

        /** Demonstrates some basic types that you can use as parameters  */
        int basicTypes(int anInt, byteArray[] aByteArray);
}
```

Android's quick win: Services & AIDL

Compiled result:

```
package com.example.android;

public interface IRemoteService extends IInterface {
    public static class Default implements IRemoteService {
        @Override
        public boolean onTransact(int i, Parcel parcel, Parcel parcel2, int i2) throws RemoteException {
            switch (i) {
            case 1:
                int getPid_ = getPid();
                parcel2.writeNoException();
                parcel2.writeInt(getPid_);
                return true;
            case 2:
                int i1 = parcel.readInt();
                byte[] bArr = parcel.createByteArray();
                int basicTypes_ = basicTypes(i1, bArr);
                parcel2.writeNoException();
                parcel2.writeInt(basicTypes_);
```

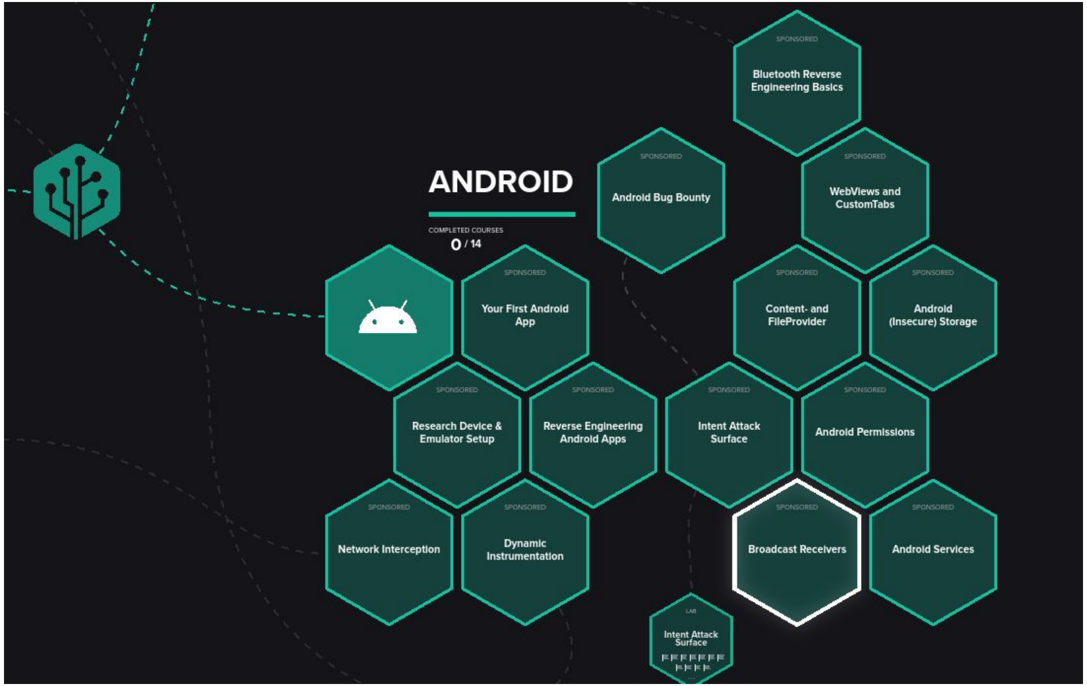
Android's quick win: getRuntime

- From the **System** class, command **getRuntime** executes provided command on the system
- A **privileged** application will have **privileged** rights for execution
- Searching for **unsecure call** to GetRuntime is always recommended

```
Process process = Runtime.getRuntime().exec("cp myFile /var/data/tmp/%s", target_name);
```

Getting deeper

- If you're new to Android and want to get deeper in, I recommend looking at Hextree.io courses, Android ones are free !



Engineer mode

Engineer mode

- **Engineer mode** are often present in IVI, allowing after-sales **analysis/debugging** of units
- It could also be used to **trigger/modify specific features** of the IVI
- To activate this mode, it may be required to:
  - Perform specific pattern on the touch screen
  - Insert an USB key with a specific file/folder
  - Use UDS command

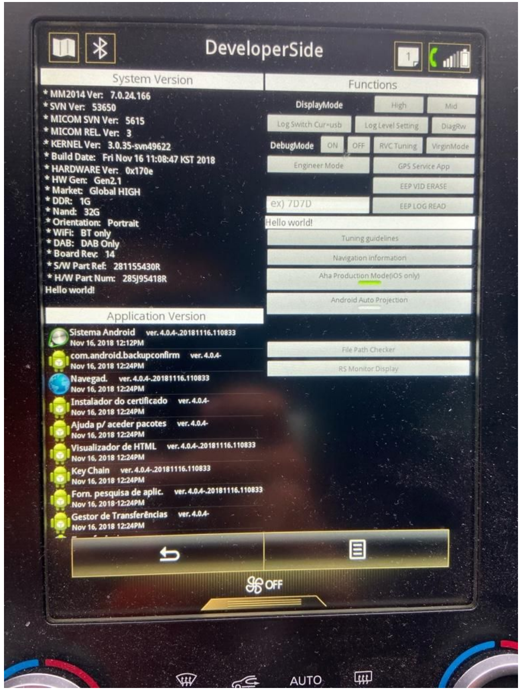


Illustration: [link] & [link]

Finding how to activate engineer mode

- For **common IVI**, method to activate engineer mode can be found on various **forums/websites**
- On Android-based IVI, search for **ClickListener** or **onLongClickListener** usage and **Intent** which could be named with ***engineer*** or ***debug***
- On some devices, using an **UDS WriteDataByIdentifier** request could activate such mode

Illustration: [link]

Lab 5 - Accessing Engineer Menu

Goals

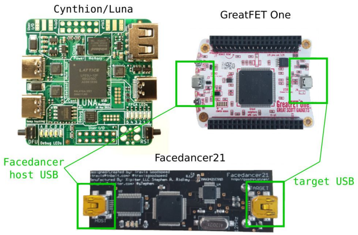
- Complete challenges **Infotainment - Engineer Mode** to access to the **Engineer Menu**

Analysing USB interface

Quarkslab

Scanning supported devices

- **IVI** may support various **USB devices**, scanning them may give us hints on some capabilities (keyboard, mass storage, usb to ethernet support, …)
- Well known device for this task is the Facedancer21 board and umap2, which are old but reliable tools
- Using the Facedancer Python library, it is simple to emulate USB devices to assess IVI with compatible hardware, like the ones below
- We recommend using board like Cynthion or the Hydradancer, which offers higher data rate and flexibility than the Facedancer21 and the GreatFET



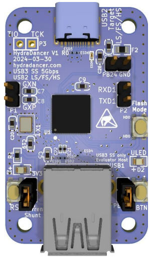
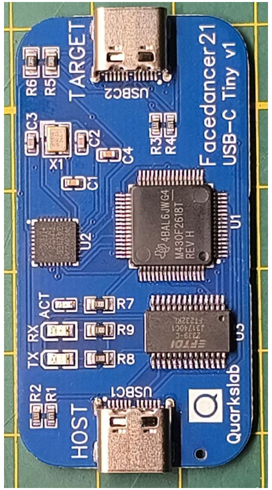


Illustration: [link] & [link]

Scanning supported devices

- **umap2** from NCC group is the go-to tool to scan supported USB devices using a **Facedancer21**
- Connect the **HOST** port of the **Facedancer21** to your computer and the **TARGET** one to the device under test
- To simply scan a USB host, we can use the following command:

```
$ umap2scan -P fd:/dev/ttyUSB0
[.. redacted for readability ..]
[INFO  ] [SmartcardDevice] Response:
260345006d0075006c006100740065006400200053006d006100720074006300610072006400
6400
[INFO  ] have been waiting long enough (over 6 secs.), disconnect
[INFO  ] [Max342xPhy] Disconnected device SmartcardDevice
[INFO  ] [Max342xPhy] Disconnect called when already disconnected
[ALWAYS] --------------------------------
[ALWAYS] Found 2 supported device(s):
[ALWAYS] 1. audio
[ALWAYS] 2. cdc_acm
```



Emulating devices

- With the Facedancer 3.0 library, it is possible to **emulate a USB device**
- Working with this feature could help to **scan for supported characteristic**, find non-filtered **Vendor ID** and **Product ID** …
- For instance, we can emulate a **mass storage device** and **display live content written in clusters** to follow dump logs without having to remove a USB stick between various attempts of an attack
- Facedancer’s Github repository holds several examples to jump into USB emulation

Emulating devices - examples

```python
from facedancer import *
from facedancer import main

@use_inner_classes_automatically
class HackRF(USBDevice): """ Device that emulates a HackRF enough to appear in ``hackrf_info``."""
    # Show up as a HackRF.
    product_string      : str = "HackRF One (Emulated)"
    manufacturer_string : str = "Great Scott Gadgets"
    vendor_id           : int = 0x1d50
    product_id          : int = 0x6089

    class DefaultConfiguration(USBConfiguration):
        class DefaultInterface(USBInterface):
            pass

    @vendor_request_handler(number=14, direction=USBDirection.IN)
    @to_device
    def handle_control_request_14(self, request):
        request.reply([2])

    @vendor_request_handler(number=15, direction=USBDirection.IN)
    @to_device
    def handle_get_version_request(self, request):
        request.reply(b"Sekret Facedancer Version")

main(HackRF)
```

Lab 6 - Exploiting USB port

Goals

- You'll analyse supported USB devices using **umap2**
- And emulate basic USB device using **Facedancer**
- Complete challenges **Infotainment - USB**

Extracting Personally Identifiable Information

Quarkslab

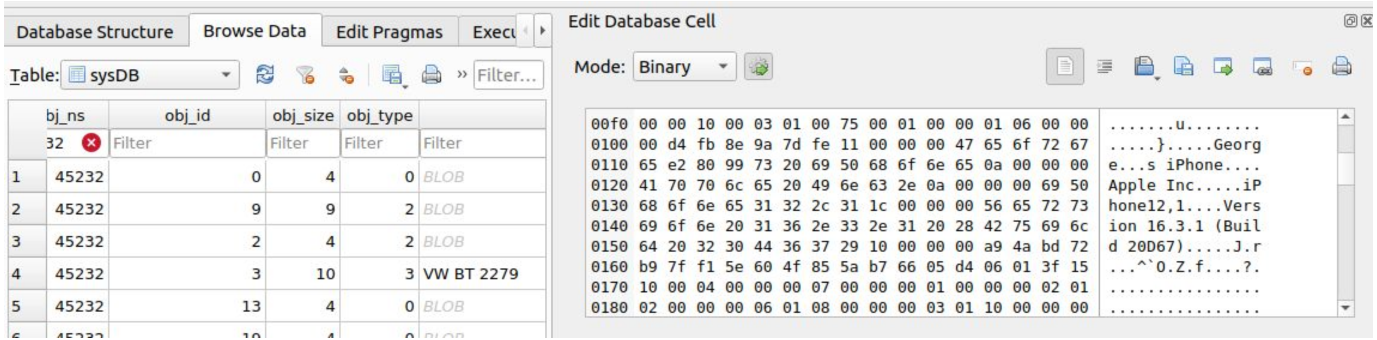
IVI: a personally identifiable information goldmine

- IVIs **store lot of data**, including:
  - Phonebook address
  - Navigation history
  - Paired phone information
  - Logs of call sent/received
  - …

- Modern **IVI** are connected to **internal/external camera**, pictures may be retrieved
- If there is some compliance with the **GDPR**, logs may still contain **privacy related data**

Illustration: [link]

IVI: a personally identifiable information goldmine




```
BEGIN:PROFILE
X-TSD-SLOT:4
X-TSD-MACADDRESS:90b123d2c60a
X-TSD-NAME:Galaxy A51
X-TSD-LRU:442
X-TSD-SORTORDER-LASTNAME-COMMA-FIRSTNAME:
X-TSD-PICTURE-VISIBLE:
X-TSD-PRIVATEPROFILE:
END:PROFILE
BEGIN:VCARD
VERSION:3.0
FN:PAU
X-TSD-ORIGIN:ME
N:;JOHN;;;;
TEL;TYPE=CELL:+32453271723
END:VCARD
BEGIN:VCARD
VERSION:3.0
FN:John, Doe
X-TSD-ORIGIN:ME
N:;John;;;;
TEL;TYPE=CELL:5749
END:VCARD
```

```
14:56:00.878    [Warn] [iMX6.Organizer.CallStack]  CallStack::addCall. Reached the maximal nr of calls. Call details, normalNumer: 4 origin: 0 ent
14:56:00.408    [Info] [J5e.Radio.AUDIO]   IMX Input DMA: iMaxAnn1 = 0x0000
14:56:00.886    [Warn] [iMX6.Organizer.CallStack]  CallStack::addCall. Reached the maximal nr of calls. Call details, normalNumer: 4586359 origin:
14:56:00.409    [Info] [J5e.Radio.AUDIO]   IMX Input DMA: iMaxAnn2 = 0x0000
14:56:00.895    [Info] [iMX6.Navi.tsd.nav.core.navlocresolver.GeoPositionResolver] resolved country: France
14:56:00.411    [Info] [J5e.Radio.AUDIO]   IMX Input DMA: iMaxAnn3 = 0x0000
14:56:00.909    [Info] [iMX6.Navi.tsd.nav.core.navlocresolver.GeoPositionResolver] resolved city: Marseille
14:56:00.412    [Info] [J5e.Radio.FmSingle-ALL]    ESC[0mQuali - Antenne 1|F:    103200kHz||PI:  -2|QS: 0|QA: 0|QSmin:30|RDS:0|Seek:0ESC[0m
14:56:00.909    [Info] [iMX6.Navi.tsd.nav.core.navlocresolver.GeoPositionResolver] cityRefinement: Bouches-du-Rhône, Provence-Alpes-Côte d'Azur
```

Collecting personal data

- Some IVI dump logs on **USB mass storage** device or **SD card**
- A specific folder or file may be required, or an activation through the engineer menu
- Trying **USB to Ethernet adapter** on USB port may lead access to **exposed services**
- It is common to have a USB **Vendor ID** or **Product ID** filtered for **USB to Ethernet adapter**, but using an **ASIX AX88772B** device it can be altered
- Scanning UDS **ReadDataByIdentifier** may also provide sensitive data

Altering Asix VID/PID

- EEPROM memory of **Asix AX88772** adapter can be flashed to modify their **Vendor ID** and **Product ID**
- It can be done using **ethtool**

```
#!/bin/bash
# Original VID:PID: 0b95:7720
# Product: AX88772
# Manufacturer: ASIX Elec. Corp.

ETH=$(ip -o link|awk -F': ' '/ether.*00:6f:00/{print $2}')
if [ "$ETH" != "" ]; then
  # VID
  sudo ethtool -E $ETH magic 0xdeadbeef offset 0x0048 length 0x01 value 0x95
  sudo ethtool -E $ETH magic 0xdeadbeef offset 0x0049 length 0x01 value 0x0B
  # PID
  sudo ethtool -E $ETH magic 0xdeadbeef offset 0x004A length 0x01 value 0x77
  sudo ethtool -E $ETH magic 0xdeadbeef offset 0x004B length 0x01 value 0x20

  sudo ethtool -e $ETH offset 0x48 length 4
fi
```

Altering Asix VID/PID

- EEPROM memory of **Asix AX88772** adapter can be flashed to modify their **Vendor ID** and **Product ID**
- It can be done using **ethtool**

```
#!/bin/bash
# Original VID:PID: 0b95:7720
# Product: AX88772
# Manufacturer: ASIX Elec. Corp.

ETH=$(ip -o link|awk -F': ' '/ether.*00:6f:00/{print $2}')
if [ "$ETH" != "" ]; then
  # VID
  sudo ethtool -E $ETH magic 0xdeadbeef offset 0x0048 length 0x01 value 0x95
  sudo ethtool -E $ETH magic 0xdeadbeef offset 0x0049 length 0x01 value 0x0B
  # PID
  sudo ethtool -E $ETH magic 0xdeadbeef offset 0x004A length 0x01 value 0x77
  sudo ethtool -E $ETH magic 0xdeadbeef offset 0x004B length 0x01 value 0x20

  sudo ethtool -e $ETH offset 0x48 length 4
fi
```

Lab 7 - Export personal data

Goals

- Exploiting **Engineer Mode** and **USB emulation**, you'll practice on recovering Personally Identifiable data stored on the IVI
- Complete challenges **Infotainment - Personally Identifiable Data**

Exploiting Bluetooth' trust

Quarkslab

Bluetooth Link Key

- **In-Vehicle Infotainment** supports Bluetooth BR/EDR to pair **smartphone** for phone calls or playing music

- When two devices paired, they exchange their **I/O capabilities** to define how to establish an encrypted connection:
  - **Just Works**
  - **Numeric comparaison**
  - **Passkey entry**
  - **Out of Band**

- Before **Bluetooth 4.1**, a **Link Key** is derived from the Pin code exchanged

- With **Bluetooth 4.1+**, the **Link Key** is derived using elliptic curves

- Communication between the two devices are encrypted based on derived value of the **Link Key**

- To **bond** two devices, they each **store** the computed **link key** for automatic pairing

Bluetooth Link Key exploitation

- Knowing the **Link Key** and the **Bluetooth address** (**BDADDR**) of a device, an attacker can mimic it to automatically connect to the target device
- In Linux, this information is stored in **/var/lib/bluetooth/**
- At the root level, there is one folder by **BDADDR** of Bluetooth adapter connected to the computer
- Each folder contains sub-folders, named also with the **BDADDR** of connected devices
- In the **info** file (if present) there will be several information, including the **Link Key** if devices were bonded

```
root@carhack:/var/lib/bluetooth# tree
.
├── 8C:68:8B:82:90:2E
│   ├── B8:27:EB:69:B6:87
│   │   ├── attributes
│   │   └── info
│   ├── cache
│   │   └── B8:27:EB:69:B6:87
│   └── settings
└── E8:65:38:47:4E:88
    └── settings

5 directories, 5 files
```

Mimicking Bluetooth address

- Some adapters can be programmed to modify their **BDADDR**
- **Cambridge Silicon Radio CSR4.**0 dongle is well known, but only supports Bluetooth 4.0
- Linux **Bluez** provides command **bdaddr** to modify the Bluetooth Address of supported chips

Bluetooth basis: getting BDADDR and scanning

- To get the Bluetooth address of the provided Bluetooth dongle, you can use the **hcitool** command
- To scan **Bluetooth BR/EDR** advertisement frames

```
$ sudo hciconfig
hci0:    Type: Primary  Bus: USB
         BD Address: 00:1A:7D:DA:71:13    ACL MTU: 679:8    SCO MTU: 48:16
         DOWN
[...]
$ sudo hciconfig hci0 up
$ sudo hcitool scan
Scan …
49:B2:CA:1A:B0:A1 (unknown)
49:B2:CA:1A:B0:A1 (unknown)
```

Bluetooth Link Key info file

- To allow your computer to use a known **Link Key**, you must in */var/lib/bluetooth* find the folder with the **bdaddr** of the adapter you want to use
- In this folder, create a new folder with the **bdaddr** of your target
- Create an empty **attribute** file
- Create an **info** file, containing at least:

```
$ cat info
[General]
Name=IVI
SupportedTechnologies=BR/EDR;

[LinkKey]
Key=DF28E75F341723D2ECA4A4B905B24F42
Type=8
PINLength=0
```

Enabling connection

- To apply changes, bluetooth system must be restarted
- Using bluetoothctl, connection to the target device could be started by typing

```
$ sudo systemctl restart bluetooth

$ sudo bluetoothctl

Agent registered
[CHG] Controller AA:BB:CC:DD:EE:FF Pairable: yes
[bluetooth]# connect 00:11:22:33:44:55
```

Lab 8 - Exploit Bluetooth Link Key

Goals

- Using the retrieved **Sqlite database**, you'll find the link key of a device
- The goal is to mimic it to establish a **Bluetooth connection** to your **IVI**
- As the provided **Bluetooth** dongle is not supported by **Bluez's bdaddr**, your dongle BDADDR will be set in the **IVI** using UDS command
- Complete challenges **Infotainment - Bluetooth Link Key**

Thank you

Contact information:

Email: contact@quarkslab.com

Phone: +33 1 58 30 81 51

Website: www.quarkslab.com

@quarkslab

Quarkslab