Security analysis of Bluetooth Low Energy Over-The-Air firmware updates

Huy Hung LE

Supervisors: Aurélien Francillon and Romain Cayre

June 29, 2023

Contents

Introduction

Device Firmware Update (DFU) Over-The-Air (OTA) is a process that allow to update firmware running on a device over a Bluetooth Low Energy (BLE) connection from a smartphone or a computer. Nordic Semiconductor provides a proprietary DFU process which implemented in many smartwatches and smartlocks as a BLE Generic Attribute (GATT) Service (see Section 1.3). The chip that are widely used are nRF51 and nRF52 (see Section 1.1). This project goal is to reverse engineer this DFU OTA process and identify potential vulnerabilities.

In Section 1, we will describe the Development Kits (DK) nRF51 and nRF52 (Section 1.1), the Software Development Kits (SDK) (Section 1.2) and an overview of a GATT Service (Section 1.3).

In Section 2, we will describe the process of doing DFU over BLE. We give some remarks about the problems may encountered when doing DFU on the DKs: Remark 3, 4, 5, 6, 7.

In Section 3, we will describe in detail the content of a firmware packet.

In Section 4, we will describe how we reverse the DFU process: capture the packets OTA (Section 4.1), the detail of DFU protocol (Section 4.2.1 and Section 4.2.2). Then we redo the DFU process by writing a python program (Section 4.4). We also gather some information about the process of validating a firmware packages in Section 4.3. Section 4.5 can be consider as an application of Section 4.4. It will be useful when we discuss about the security in Section 5.

Section 5 contains all the important result related to security of the DFU process.

Finally, I would like to thank my supervisors Aurélien Francillon and Romain Cayre. I would not able to finish this project without them. I also learn a lot of interesting stuffs when doing this project: how to reverse engineering a protocol, understanding BLE, experiment with DKs, using tools and fixes the problems,...

1 Preliminaries

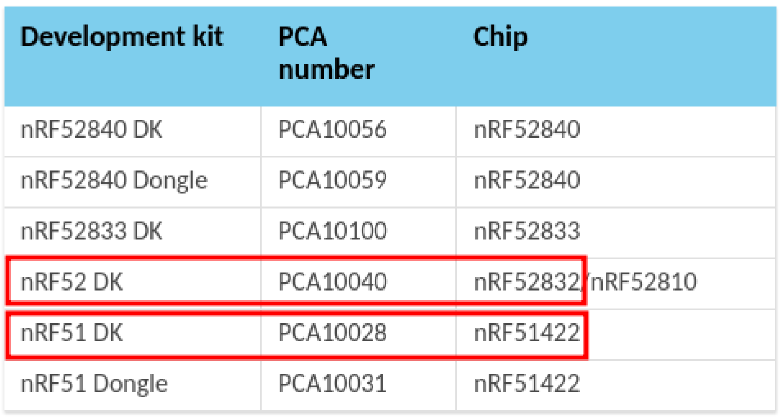
In this section, we will describe the DKs (Section 1.1), SDKs (Section 1.2) which are used in this project. We also give a few works on GATT Services of BLE in Section 1.3.

1.1 Development kit

In this project, we will use two DKs from Nordic Semiconductor:

- nRF51, see https://www.nordicsemi.com/Products/Development-hardware/nrf51-dk.
- nRF52, see https://www.nordicsemi.com/Products/Development-hardware/nrf52-dk.

The table below shows the relation between DKs, PCA numbers and chips.



Development kit	PCA number	Chip
nRF52840 DK	PCA10056	nRF52840
nRF52840 Dongle	PCA10059	nRF52840
nRF52833 DK	PCA10100	nRF52833
nRF52 DK	PCA10040	nRF52832/nRF52810
nRF51 DK	PCA10028	nRF51422
nRF51 Dongle	PCA10031	nRF51422

Figure 1: Relation between development kits, PCA numbers, and chips
Source:
https://infocenter.nordicsemi.com/topic/ug_gsg_ses/UG/gsg/chips_and_sds.html

Remark 1. The PCA number will be encountered a lot when we explore the software development kits: pca10028 for the nrf51 and pca10040 for nrf52.

1.2 Software development kit

The SDKs used are the nRF5 SDK, range from SDK v9.0.0 (2015) to v17.1.0 (2021). From now on, we will refer to them simply as "SDK". They can be downloaded at `https://www.nordicsemi.com/Products/Development-software/nRF5-SDK/Download?lang=en#infotabs`.

Remark 2. Here is some further information about SDKs:

- The SDK v9.0.0 and v10.0.0 are called nRF51 SDK.
- The SDK v12.3.0 is the latest version that support nRF51.
- We can only download the SDK v7.0.0 and from v9.0.0 to v17.1.0. The structure of SDK v7.2.0 is a bit different. Consequently, SDK v7.2.0 and earlier version are not subjected to analysis in this project.
- Currently the nRF5 SDK is in maintenance mode. Nordic Semiconductor recommend the nRF Connect SDK ([6]).

1.3 Generic Attribute Service

In this section, we will provide a short overview of the Generic Attribute Profile (GATT) of BLE (for more details, see [1, chapter 4],[2, chapter 13]):

- BLE also known as Bluetooth Smart started as part of the Bluetooth Core Specification version 4.0 (see [3]). It is aimed at "ultra" low power devices.
- The GATT within BLE defines how to manage and exchange data between BLE devices.
- Within the GATT, there are services. Each service contains its own set of characteristics. These characteristics consist of attributes.
- Attribute is the smallest data entities defined by the GATT. Some common attributes are handle (16 bits), type (UUIDs), permissions, value (can be UUIDs or anything), descriptors,...

The DFU process is implemented as a BLE GATT service and a set of associated characteristics. We will discuss more about the DFU process and its corresponding GATT service in Chapter 2 and Chapter 4. Also see Section 4.5 to know the implemented services and characteristics.

2 Device Firmware Update process

A Softdevice is a precompiled and linked binary software implementing a wireless protocol developed by Nordic Semiconductor.

Device Firmware Update is the process of update the SoftDevice, bootloader, application or its combination on a device called the DFU target (for example, DK nRF51 and DK nRF52).
The new SoftDevice, bootloader, or application image can be transferred over-the-air (OTA) using the Nordic BLE DFU Service by a device called the DFU controller (for example, Android phone, Apple phone, laptop,...)

There are two main generation of the DFU process: The DFU process with open bootloader (SDK v11.0.0 and earlier) and the DFU process with secure bootloader (SDK v12.1.0 and later).

In this section, we will describe some main steps to do the DFU process and have some remarks.

2.1 Compiling and flashing files from SDK to the DK

In the gitlab repository ([4]) of this project, we have some compiled files from each SDK version. We also have some Makefile files to have easier to flashing files.

Some SDKs also provide pre-compiled files to flash to the DKs. In this section, we will explain the process of compiling and flashing files without using the pre-compiled from SDKs.

2.1.1 Compiling files

We will use `gcc-arm-none-eabi` as a compiler to compile program in the SDK. Each SDK version will need a specific version of `gcc-arm-none-eabi`. In this project, we use the extract version of `gcc-arm-none-eabi` for each SDK version. See Remark 3 if we want to use the latest version.

Below is the main steps to compile a program. Skip step 2 and 3 if we want to flash `open bootloader`.

1. Modify the file `<SDK-folder>/components/toolchain/gcc/Makefile.posix`. Here is an example of the original content of this file:

```
GNU_INSTALL_ROOT := /usr/bin/gcc-arm-none-eabi-4_8-2014q1
GNU_VERSION := 4.8.3
GNU_PREFIX := arm-none-eabi
```

We modify the first line: replace `/usr/bin` by the path to the extracted folder of `gcc-arm-none-eabi` after downloading.

2. Install `micro-ecc`:

```
cd <SDK-folder>/external/micro-ecc
git clone https://github.com/kmackay/micro-ecc.git .
```

Then go to each folder `nrf52hf_armgcc/armgcc` and `nrf52nf_armgcc/armgcc`, run `make`.

3. Create a pair of public key and private key using `nrfutil` (https://infocenter.nordicsemi.com/topic/sdk_nrf5_v17.1.0/lib_bootloader_dfu_keys.html).
Store public key in a file named `dfu\_public\_key.c`. We replace the old file `dfu_public_key.c` in the SDK with the newly created file. The old file often is in the folder:

- SDK v12.3.0 and earlier: `<SDK-folder>/examples/dfu/bootloader_secure/`
- SDK v13.0.0-v14.2.0: `<SDK-folder>/examples/dfu/dfu_req_handling`
- SDK v15.0.0-v17.1.0: `<SDK-folder>/examples/dfu/`

The private key `*.pem` will be use later when we create signed firmware packages.

4. Now, we have setup the requirement to compile the bootloader or application that we want in the SDK: Go to the folder of an example, choose the correct pca number (see Figure 1): pca10028 for nrf51 and pca10040 for nrf52, go to the folder armgcc and run `make`.

Remark 3. We can use the latest version of gcc-arm-none-eabi. However we need some additional actions when compiling a program:

- In step 1, we need to modify the version in second line to be the latest version. For the path, modify all the path (not only `/usr/bin`) to match the path of the binary file of the latest version.
- In step 4, we need to search and remove the option `-Werror` in the `Makefile` before running `make`.

2.1.2 Flashing files

There are two ways:

- When connect the board to the laptop, the board will appear as a folder name JLINK. We can simply copy the hex file to this folder to flash this file to the board.
- The second way is to use `nrfjprog`. In this project, we use this ways to flash files to the board.

Remark 4. About using `nrfjprog`, if we see the Makefile from the SDK, some SDK use different options such as --reset, --sectorerase. Some options works with a SDK but not with other SDKs. For better experience, we recommend:

- Use --reset option in a separate command
- Do not use the option --sectorerase. Instead, we erase the board first then flash the file.

2.2 Creating firmware packets

To create a firmware packets, we need to use `nrfutil`, available at https://www.nordicsemi.com/Products/Development-tools/nrf-util. Some SDKs also provide pre-created firmware packages to test.

Remark 5. • To create firmware packages for doing DFU on board with open bootloader, we need to use a very old version of `nrfutil` (v0.5.1) which is a little bit tricky to find and install.

• We will refer old-nrfutil to this very old version of nrfutil and the newer version is simply called nrfutil.

• Below we explain the steps to install the old-nrfutil.

First we need to create a python environment with python version 2.7. We use Conda to do this. Suppose that the name of the environment is py27:

```
conda create --name py27 python=2.7
```

Then we activate the python2.7 environment:

```
conda activate py27
```

Select a folder to clone the repository of pc-nrfutil (Public archive) and checkout a branch named `0_5_1`:

```
git clone https://github.com/NordicSemiconductor/pc-nrfutil .
git checkout 0_5_1
```

The follow the instruction in this git branch to install old-nrfutil. The old-nrfutil only works when we activate the py27 environment. To deactivate this environment to use the new nrfutil, we run `conda deactivate`.

Remark 6. We have some observations when creating the firmware packages:

• For `open bootloader`, we only need to specify the version of application or bootloader or softdevice that we want to create the firmware. Other version information will be the default value if not specify.

• For `secure bootloader`, we must specify the option --hw-version (hardware version), --sd-req (softdevice required on the board). The hardware version is 51 or 52. The sd-req value is specific in the nrfutil tool when using --help option.

2.3 Doing the DFU process

We can use nrfutil to do the DFU process from laptop via BLE, usb-serial,... However, in this project, we use an application named nrfConnect in Android or Iphone.

To do the DFU process, first, send the firmware that we created to smart phone device. Now, open the application nrfConnect and scan. After flashing files to the board, we should see the advertiser name "DfuTarg". Choose to do the DFU and select the firmware. It should show a table of data rate and the process doing DFU. When it reaches 100%, the DFU should success.

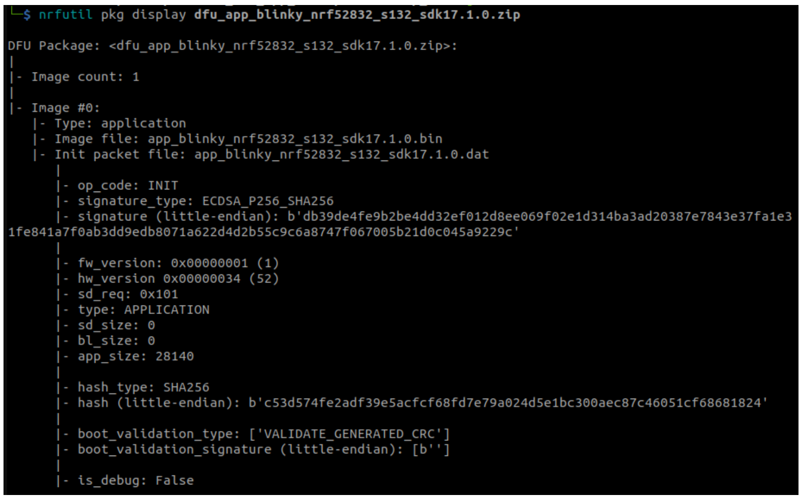
Remark 7. We recommend to use Android phone for some reasons:

- the nrfConnect application on Android has more functionality than in Iphone.
- Android phone has the ability to record the bluetooth packets sent from it. It will be useful when we want to analysis the DFU protocol. Iphone may also have similar ability but we have not found yet.

3 Reverse the firmware packets

In this section, we describe the content of the firmware packet and see the difference between two cases: open bootloader and secure bootloader.

The firmware package is a zip file. We can use nrfutil to read a firmware packet create by nrfutil (not works with old-nrfutil).



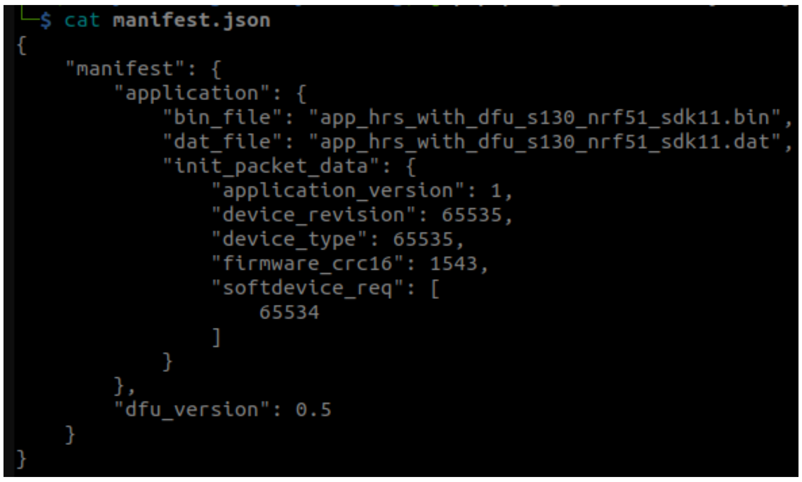
```
└─$ nrfutil pkg display dfu_app_blinky_nrf52832_s132_sdk17.1.0.zip

DFU Package: <dfu_app_blinky_nrf52832_s132_sdk17.1.0.zip>:
|
|- Image count: 1
|
|- Image #0:
   |- Type: application
   |- Image file: app_blinky_nrf52832_s132_sdk17.1.0.bin
   |- Init packet file: app_blinky_nrf52832_s132_sdk17.1.0.dat
      |
      |- op_code: INIT
      |- signature_type: ECDSA_P256_SHA256
      |- signature (little-endian): b'db39de4fe9b2be4dd32ef012d8ee069f02e1d314ba3ad20387e7843e37fa1e3
1fe841a7f0ab3dd9edb8071a622d4d2b55c9c6a8747f067005b21d0c045a9229c'
      |
      |- fw_version: 0x00000001 (1)
      |- hw_version 0x00000034 (52)
      |- sd_req: 0x101
      |- type: APPLICATION
      |- sd_size: 0
      |- bl_size: 0
      |- app_size: 28140
      |
      |- hash_type: SHA256
      |- hash (little-endian): b'c53d574fe2adf39e5acfcf68fd7e79a024d5e1bc300aec87c46051cf68681824'
      |
      |- boot_validation_type: ['VALIDATE_GENERATED_CRC']
      |- boot_validation_signature (little-endian): [b'']
      |
      |- is_debug: False
```

Figure 2:

After extracting the package, we have three files: two files with same name but different extension `.dat`, `.bin` and a file `manifest.json`.

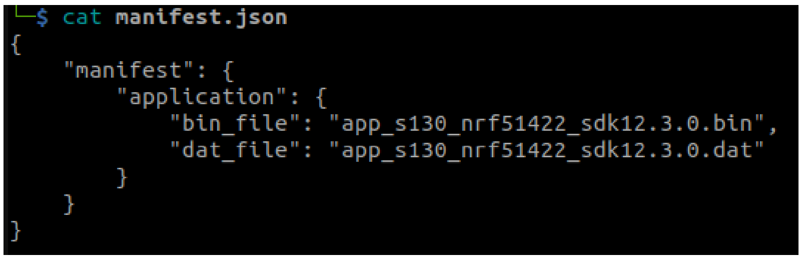
3.1 File manifest.json

Let us look at the files `manifest.json` in open bootloader case and secure bootloader case:



```
└─$ cat manifest.json
{
    "manifest": {
        "application": {
            "bin_file": "app_hrs_with_dfu_s130_nrf51_sdk11.bin",
            "dat_file": "app_hrs_with_dfu_s130_nrf51_sdk11.dat",
            "init_packet_data": {
                "application_version": 1,
                "device_revision": 65535,
                "device_type": 65535,
                "firmware_crc16": 1543,
                "softdevice_req": [
                    65534
                ]
            }
        },
        "dfu_version": 0.5
    }
}
```

Figure 3: manifest.json - open bootloader



```
└─$ cat manifest.json
{
    "manifest": {
        "application": {
            "bin_file": "app_s130_nrf51422_sdk12.3.0.bin",
            "dat_file": "app_s130_nrf51422_sdk12.3.0.dat"
        }
    }
}
```

Figure 4: manifest.json - secure bootloader

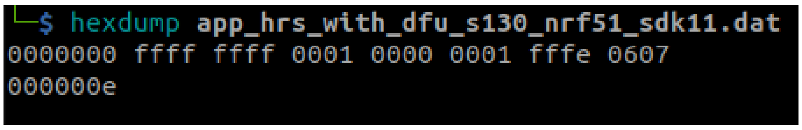
We can see that:

- Both files has a key to indicate this is a firmware of an application. They also have the name of the `bin_file` and `dat_file` in this firmware.
- The open-bootloader file has more information about the firmware. Some values such as 65535 is the default value (when we do not specify these value during creating the firmware packets).

the open bootloader provide more detail about the firmware packages

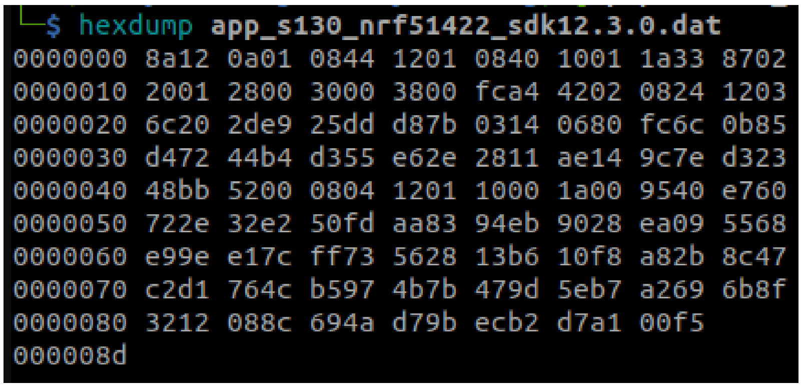
3.2 File .dat

Let us look at the file `.dat`:



```
└─$ hexdump app_hrs_with_dfu_s130_nrf51_sdk11.dat
0000000 ffff ffff 0001 0000 0001 fffe 0607
000000e
```

Figure 5: *.dat file - open bootloader



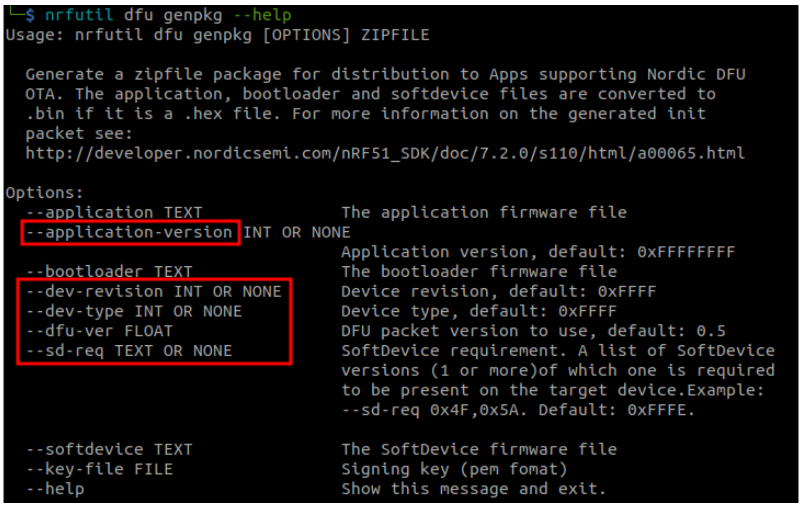
```
└─$ hexdump app_s130_nrf51422_sdk12.3.0.dat
0000000 8a12 0a01 0844 1201 0840 1001 1a33 8702
0000010 2001 2800 3000 3800 fca4 4202 0824 1203
0000020 6c20 2de9 25dd d87b 0314 0680 fc6c 0b85
0000030 d472 44b4 d355 e62e 2811 ae14 9c7e d323
0000040 48bb 5200 0804 1201 1000 1a00 9540 e760
0000050 722e 32e2 50fd aa83 94eb 9028 ea09 5568
0000060 e99e e17c ff73 5628 13b6 10f8 a82b 8c47
0000070 c2d1 764c b597 4b7b 479d 5eb7 a269 6b8f
0000080 3212 088c 694a d79b ecb2 d7a1 00f5
000008d
```

Figure 6: *.dat file - secure bootloader

We can see that in case of open bootloader, the content of dat file is much more simpler than the case of secure bootloader.

3.2.1 File .dat - open bootloader

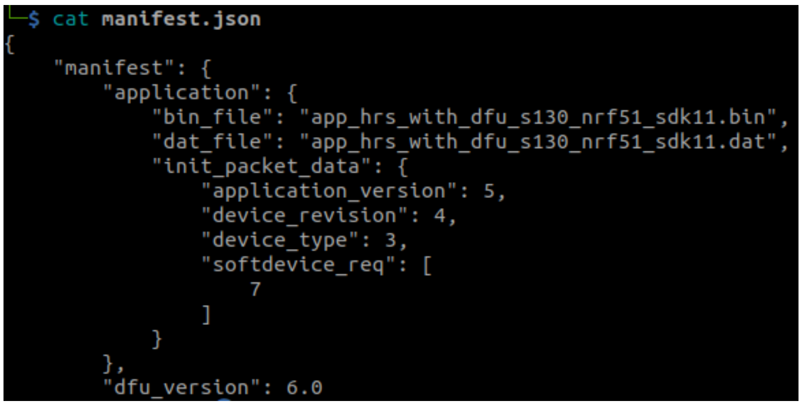
Now, by creating many firmware packages with different options values, we can identify the meaning of some part of this dat file. Let us recall some option in old-nrfutil



```
└─$ nrfutil dfu genpkg --help
Usage: nrfutil dfu genpkg [OPTIONS] ZIPFILE

  Generate a zipfile package for distribution to Apps supporting Nordic DFU
  OTA. The application, bootloader and softdevice files are converted to
  .bin if it is a .hex file. For more information on the generated init
  packet see:
  http://developer.nordicsemi.com/nRF51_SDK/doc/7.2.0/s110/html/a00065.html

Options:
  --application TEXT              The application firmware file
  --application-version INT OR NONE
                                  Application version, default: 0xFFFFFFFF
  --bootloader TEXT               The bootloader firmware file
  --dev-revision INT OR NONE      Device revision, default: 0xFFFF
  --dev-type INT OR NONE          Device type, default: 0xFFFF
  --dfu-ver FLOAT                 DFU packet version to use, default: 0.5
  --sd-req TEXT OR NONE           SoftDevice requirement. A list of SoftDevice
                                  versions (1 or more)of which one is required
                                  to be present on the target device.Example:
                                  --sd-req 0x4F,0x5A. Default: 0xFFFE.

  --softdevice TEXT               The SoftDevice firmware file
  --key-file FILE                 Signing key (pem fomat)
  --help                          Show this message and exit.
```

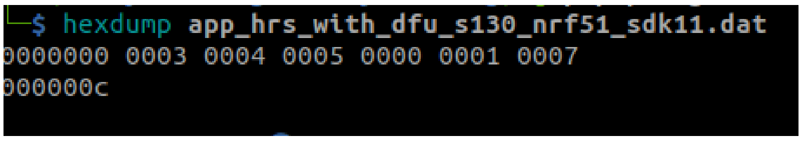
Figure 7: Old-nrfutil options

Now, we create a firmware with all 5 options dev-type = 3, dev-revision = 4, application-version = 5, dfu-ver =6, sd-req = 7:



```
└─$ cat manifest.json
{
    "manifest": {
        "application": {
            "bin_file": "app_hrs_with_dfu_s130_nrf51_sdk11.bin",
            "dat_file": "app_hrs_with_dfu_s130_nrf51_sdk11.dat",
            "init_packet_data": {
                "application_version": 5,
                "device_revision": 4,
                "device_type": 3,
                "softdevice_req": [
                    7
                ]
            }
        },
        "dfu_version": 6.0
```

Figure 8: dev-type = 3, dev-revision = 4, application-version = 5, dfu-ver =6, sd-req = 7

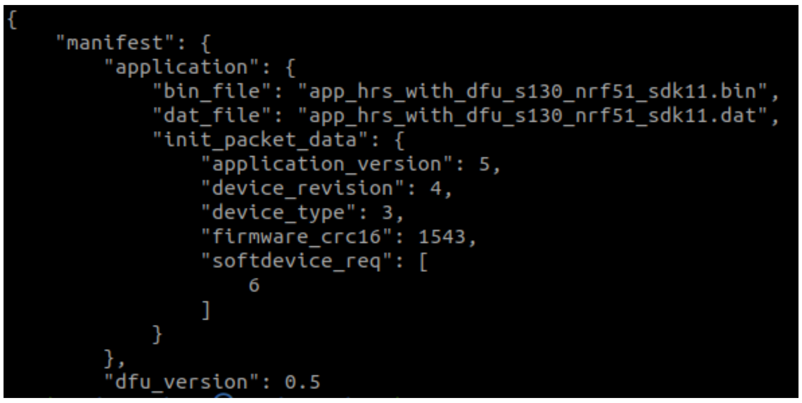


```
└─$ hexdump app_hrs_with_dfu_s130_nrf51_sdk11.dat
0000000 0003 0004 0005 0000 0001 0007
000000c
```

Figure 9: dev-type = 3, dev-revision = 4, application-version = 5, dfu-ver =6, sd-req = 7

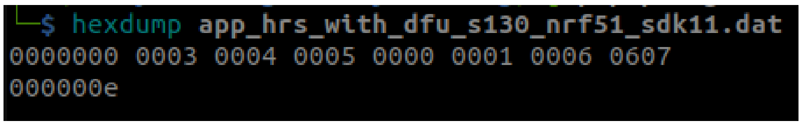
We can see that: first two byte is dev-type, next is dev-revision, application-version. The next four bytes is unknown. The last two bytes is sd-req.

However, if we does not specify dfu-version: dfu-type = 3, dev-revision = 4, application-version = 5 , sd-req = 6:



```
{
    "manifest": {
        "application": {
            "bin_file": "app_hrs_with_dfu_s130_nrf51_sdk11.bin",
            "dat_file": "app_hrs_with_dfu_s130_nrf51_sdk11.dat",
            "init_packet_data": {
                "application_version": 5,
                "device_revision": 4,
                "device_type": 3,
                "firmware_crc16": 1543,
                "softdevice_req": [
                    6
                ]
            }
        },
        "dfu_version": 0.5
```

Figure 10: dfu-type = 3, dev-revision = 4, application-version = 5 , sd-req = 6



```
└─$ hexdump app_hrs_with_dfu_s130_nrf51_sdk11.dat
0000000 0003 0004 0005 0000 0001 0006 0607
000000e
```

Figure 11: dfu-type = 3, dev-revision = 4, application-version = 5 , sd-req = 6

We can see that the last two bytes is CRC (0x0607=1543), two bytes before is the sd-req. The bytes 0000 0001 is unknown.

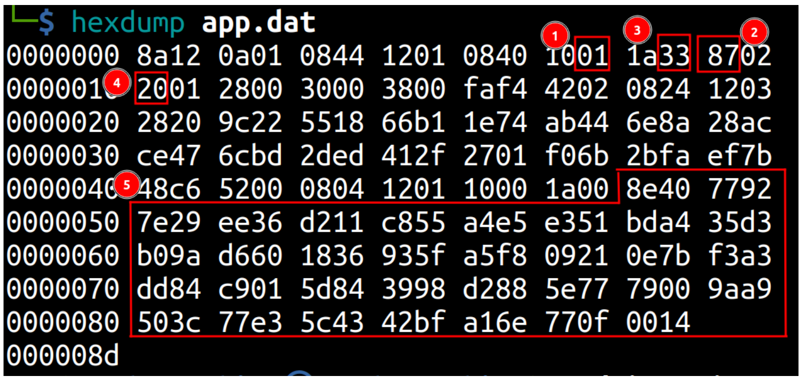
In conclusion, we guess that:

- the first two bytes is the device type
- next two bytes is the device revision
- next two bytes is the application version
- next four bytes seems to be fixed: 0000 0001
- next two bytes is the softdevice required
- the dat file is 12 bytes or 14 bytes. If the dfu-version is specified, the dat file is only 12 bytes. If not, the dat file is 14 bytes and the last two bytes is CRC

Remark 8. We also want to mention a link `https://infocenter.nordicsemi.com/topic/com.nordic.infocenter.sdk5.v11.0.0/bledfu_example_init.html`. This link explain the content of the init package (dat file): two bytes for device type, two bytes for device revision, but four bytes for the application verions, two bytes for supported softdevice, two bytes for checksum CRC-16-CCIT (which is optional). However, as we analyze above, it seems confused.

3.2.2 File .dat - secure bootloader

By doing similar as above, we know some information from the file `.dat`:

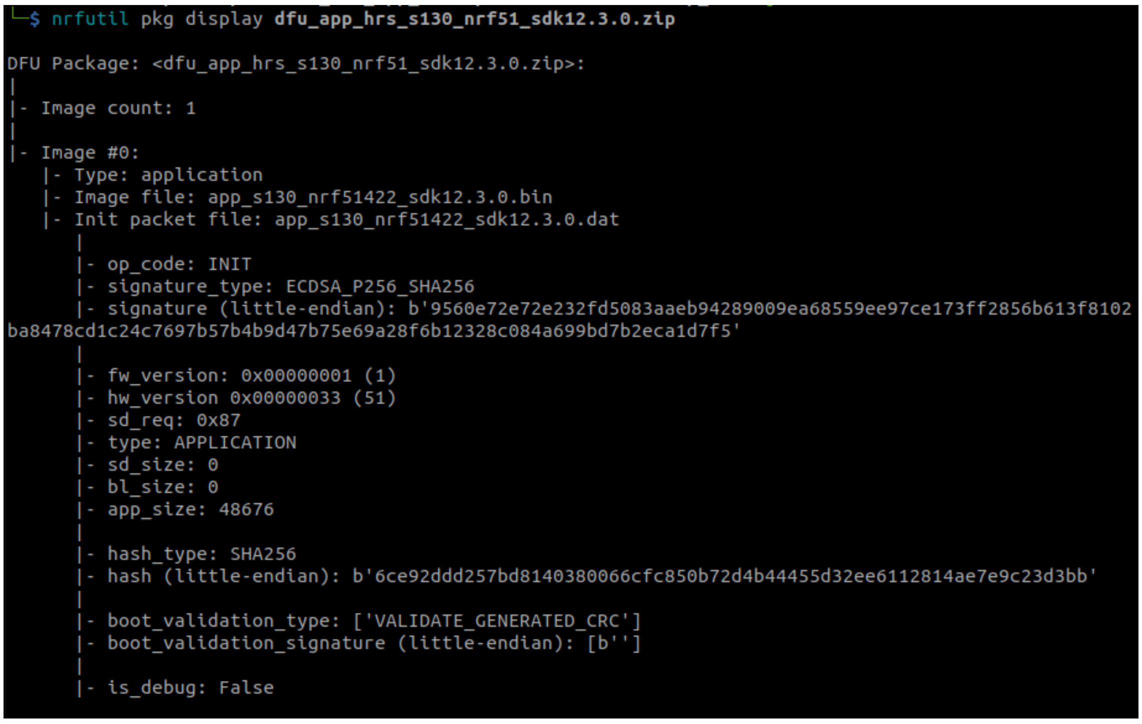


```
└─$ hexdump app.dat
0000000 8a12 0a01 0844 1201 0840 1001 1a33 8702
0000010 2001 2800 3000 3800 faf4 4202 0824 1203
0000020 2820 9c22 5518 66b1 1e74 ab44 6e8a 28ac
0000030 ce47 6cbd 2ded 412f 2701 f06b 2bfa ef7b
0000040 48c6 5200 0804 1201 1000 1a00 8e40 7792
0000050 7e29 ee36 d211 c855 a4e5 e351 bda4 35d3
0000060 b09a d660 1836 935f a5f8 0921 0e7b f3a3
0000070 dd84 c901 5d84 3998 d288 5e77 7900 9aa9
0000080 503c 77e3 5c43 42bf a16e 770f 0014
000008d
```

Figure 12: Reverse dat file - secure bootloader

- 1: application-version
- 2: sd-req
- 3+4 may related to hardware (we haven't check carefully)
- 5: signature of the hash.

Remark 9. We can also see the information of a firmware zip file by nrfutil. But this only works for secure bootloader firmware packages.



```
└─$ nrfutil pkg display dfu_app_hrs_s130_nrf51_sdk12.3.0.zip

DFU Package: <dfu_app_hrs_s130_nrf51_sdk12.3.0.zip>:
|
|- Image count: 1
|
|- Image #0:
   |- Type: application
   |- Image file: app_s130_nrf51422_sdk12.3.0.bin
   |- Init packet file: app_s130_nrf51422_sdk12.3.0.dat
      |
      |- op_code: INIT
      |- signature_type: ECDSA_P256_SHA256
      |- signature (little-endian): b'9560e72e72e232fd5083aaeb94289009ea68559ee97ce173ff2856b613f8102
ba8478cd1c24c7697b57b4b9d47b75e69a28f6b12328c084a699bd7b2eca1d7f5'
      |
      |- fw_version: 0x00000001 (1)
      |- hw_version 0x00000033 (51)
      |- sd_req: 0x87
      |- type: APPLICATION
      |- sd_size: 0
      |- bl_size: 0
      |- app_size: 48676
      |
      |- hash_type: SHA256
      |- hash (little-endian): b'6ce92ddd257bd8140380066cfc850b72d4b44455d32ee6112814ae7e9c23d3bb'
      |
      |- boot_validation_type: ['VALIDATE_GENERATED_CRC']
      |- boot_validation_signature (little-endian): [b'']
      |
      |- is_debug: False
```

Figure 13:

3.3 File .bin

This file is actually data of the hex file application, bootloader or softdevice.

4 Reverse engineer DFU protocol

A common method is to capture the file sent over the air then analyze all the data exchanged. We will explain how to sniff the data in Section 4.1 since it is not trivial with BLE. Then, we analyze the protocol in Section 4.2. And finally, in Section 4.4, with the analyzing result, we write a python program to do the DFU process without application nrfConnect on smart phone.

4.1 Captures the packages over the air

There are two ways. However, the most stable way is using an Android phone.

4.1.1 Captured the packets using Wireshark and another DK

First we need another DK as an interface to capture the packet. We need to flash a specific firmware (nRF Sniffer) to the DK then configure Wireshark to be able to receive packets of the DFU process over the air. See `https://www.nordicsemi.com/Products/Development-tools/nrf-sniffer-for-bluetooth-le` for more details.

Remark 10. It seems that we can use DK nrf52 to capture the DFU on nrf51 but not the inverse. The reason seems to relate to hardware support and compatibility.

4.1.2 Captured the packets using Android phone

The folder and functionality may different between Android phone but the idea are the same. Here we use "Realme Narzo 50i Prime" phone.

First, turn off Bluetooth, then Enable Bluetooth HCI snoop log on Android phone. Then turn on Bluetooth and do the DFU process. After that, Disable Bluetooth HCI snoop log.

Now, we need to generate the log, we can choose "Bug report" in Android phone and wait for it to generate the report. The report will be stored in folder `/bugreports/` in Android phone. We can retrieve this zip file by using `adb pull`.

Another way to generate the log is to use `adb bugreport` in laptop. The log will be created in the current folder.

After generating the log, the log can be found in the zip file under the folder `FS/data/misc/bluetooth/logs/` with the name `btsnoop_hci.log`. This file can be viewed normally in Wireshark.

4.2 Analyzing the captured packages

Fortunately, the SDK documentation has the information about the meaning of value sent through the air. However, it is not available in the latest SDK version. We have to find these information in some previous versions of SDK. We will not rewrite them here but provide the link for the information:

For open bootloader, these information mostly is in SDKv11.0.0 --> Examples --> DFU bootloader examples --> BLE&HCI/UART Bootloader/DFU --> Transport layers. See `https://infocenter.nordicsemi.com/topic/com.nordic.infocenter.sdk5.v11.0.0/bledfu_transport_bleservice.html`.

For secure bootloader, these information mostly is in SDKv12.3.0 --> Libraries --> Bootloader modules --> DFU transport --> BLE. See `https://infocenter.nordicsemi.com/topic/com.nordic.infocenter.sdk5.v12.3.0/lib_dfu_transport_ble.html`

Now we present the protocol of DFU on open bootloader and secure bootloader,

4.2.1 DFU protocol - open bootloader

1. Trigger the DFU process: subscribe to descriptor of Legacy DFU Control Point.
2. Start DFU process 0x01 + firmware type: application, bootloader or softdevice.
3. Send length of softdevice, bootloader, application.
4. Initialize DFU parameter: send 0x02 00.
5. Send firmware dat file. It is also called init packet.
6. Initialize DFU parameter complete: send 0x02 01.
7. Set PRN (Packet receive notification) value to 10
8. Prepare to send firmware bin file: send 0x03.
9. Send the firmware bin file.

10. Ask to validate the firmware image: send 0x04.
11. Ask to activate the firmware image: send 0x05.

4.2.2 DFU protocol - secure bootloader

1. Trigger the DFU process: subscribe to descriptor of DFU Control Point.
2. Seclect Command Object.
3. Set PRN (Packet receive notification) value to 0.
4. Create Command Object with size of firmware dat file.
5. Send the firmware dat file.
6. Request to calculate CRC.
7. Request to execute the sent file.
8. Set PRN value to 10.
9. Send the firmware bin file. This step is more complicate than the other steps. See Remark 11.

Remark 11. The step of sending firmware bin file is different with the open bootloader case. In step 9 above, to send a large file greater than 4096 bytes, the process is as follows:
1. Select last command object and create data object with size = 4096.
2. Send this file as packets of 20 bytes data until 4096 bytes.
3. Request to calculate CRC.
4. Request to execute.
5. Repeat from step 1 until send all the firmware bin file. For the last part of file which is smaller than <4096, the size in step 1 will be the size of this last part.

4.3 Validate the firmware

The process of validating the firmware is documented in some SDK versions. This section is to gather those information to be able to analyze the security in Section 5.

4.3.1 Validate the firmware - open bootloader

The information is in `https://infocenter.nordicsemi.com/topic/com.nordic.infocenter.sdk5.v11.0.0/bledfu_example_init.html`.

Procedure to validate the updated firmware image. The current implementation supports only CRC validation in addition to size validation (which is always performed). CRC validation is optional and uses the information provided in the Receive Init Data procedure.

The implementation in SDK 11.0.0 includes checks for Device type and revision, Supported SoftDevices, and the checksum, but not for the Application version.

Remark 12. As mentioned in the link, we can disable the check by not specific it when create firmware packages (hence, it will use default values such as 0xFFFF for device type and revision, 0xFFFE for softdevice)

4.3.2 Validate the firmware - secure bootloader

The information for validation is in:
SDK v12.3.0: `https://infocenter.nordicsemi.com/index.jsp?topic=%2Fcom.nordic.infocenter.sdk5.v12.3.0%2Fble_sdk_app_dfu_bootloader.html&anchor=lib_bootloader_dfu_init`
SDK v17.1.0: `https://infocenter.nordicsemi.com/topic/sdk_nrf5_v17.1.0/lib_bootloader_dfu_validation.html`

In brieft, the validation process consists of pre-validation (after send the firmware dat file) and post-validation (after sending the firmware bin file).

Checking values: Hardware version, SoftDevice version, Firmware version, the hash of the packet, Signature of the packet, Available space (or Firmware size).

The order of checking may different between SDK versions.

The acceptance rules for versions:

- Hardware version: need to match.

- Softdevice Firmware ID: need to match.
- Firmware version: some SDKs require "$>$", some SDKs require "$\geq$".

4.4 Using Python to do the DFU process

With the information from Section 4.2.1 and 4.2.2, we were able to write a python program that can do the DFU process from laptop (see [4, folder `dfu_by_python`] for the latest version). Here we use a private framework name Whad-client ([5]) from Whad-team . It is developed by Romain Cayre and his team. It helps us easier in interacting with GATT server with action such as scanning, connecting, discovering, sending Write request, Write command,...

Our python program can do:

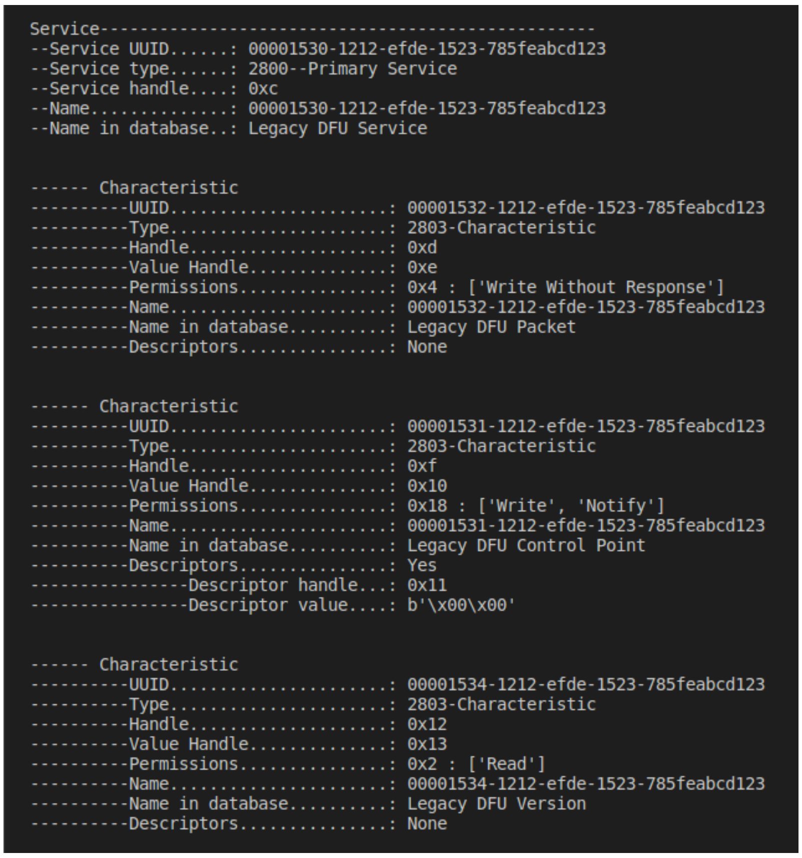
- Scan for bluetooth devices, select a device in the list and connect to this device.
- Discover the services and characteristics of this device and print its attribute handle, UUIDs, name, value,...
- We also search the UUID in a database provided by NordicSemiconductor to get the corresponding name if it is available. The database is at `https://github.com/NordicSemiconductor/bluetooth-numbers-database`.
- The mtu default value is 23. We try to set another mtu value to see if it is support by the board.
- Read the firmware zip file, unzip it; choose the correct firmware dat file and bin file from the manifest.json file, calculate the sizes and send them.

4.5 Services and Characteristics discovery - Open vs Secure bootloader

We would like to devote this section to discuss about difference of the Services and Characteristics between open and secure bootloader. Here, we use the python program from Section 4.4.

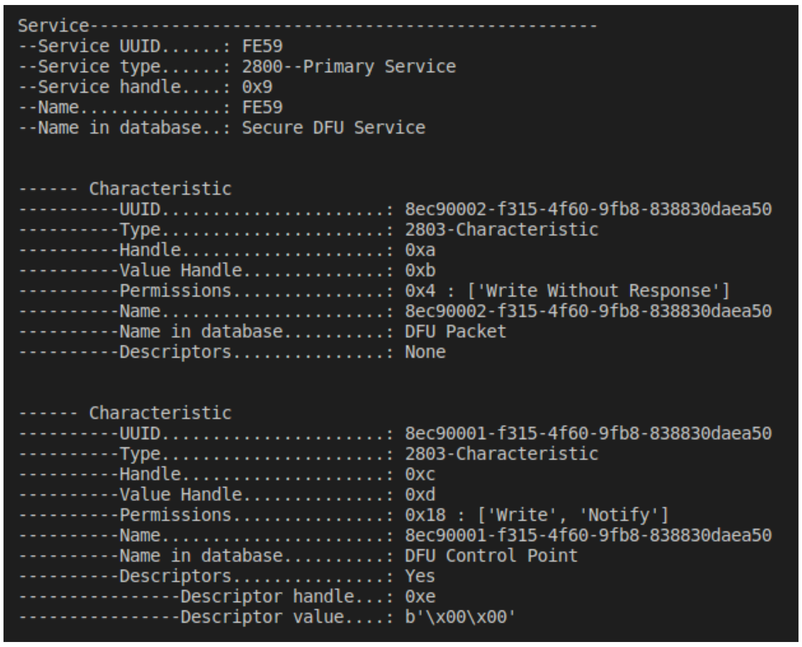
You can find in the end of this section some screenshots about GATT server when we run the python program to discover the device. Some differences are:

- Name: The open bootloader has word "Legacy" in its name.
- UUIDs:
  Open bootloader: Service UUID start: with 0x1530; Characteristic UUID start with 0x1531.
  Secure bootloader: Service UUID has short form: 0xFE59.
- Open bootloader has extra Characteristic named "Legacy DFU Version" which readable.



```
Service--------------------------------------------------
--Service UUID......: 00001530-1212-efde-1523-785feabcd123
--Service type......: 2800--Primary Service
--Service handle....: 0xc
--Name..............: 00001530-1212-efde-1523-785feabcd123
--Name in database..: Legacy DFU Service


------ Characteristic
----------UUID......................: 00001532-1212-efde-1523-785feabcd123
----------Type......................: 2803-Characteristic
----------Handle....................: 0xd
----------Value Handle..............: 0xe
----------Permissions...............: 0x4 : ['Write Without Response']
----------Name......................: 00001532-1212-efde-1523-785feabcd123
----------Name in database..........: Legacy DFU Packet
----------Descriptors...............: None


------ Characteristic
----------UUID......................: 00001531-1212-efde-1523-785feabcd123
----------Type......................: 2803-Characteristic
----------Handle....................: 0xf
----------Value Handle..............: 0x10
----------Permissions...............: 0x18 : ['Write', 'Notify']
----------Name......................: 00001531-1212-efde-1523-785feabcd123
----------Name in database..........: Legacy DFU Control Point
----------Descriptors...............: Yes
----------------Descriptor handle...: 0x11
----------------Descriptor value....: b'\x00\x00'


------ Characteristic
----------UUID......................: 00001534-1212-efde-1523-785feabcd123
----------Type......................: 2803-Characteristic
----------Handle....................: 0x12
----------Value Handle..............: 0x13
----------Permissions...............: 0x2 : ['Read']
----------Name......................: 00001534-1212-efde-1523-785feabcd123
----------Name in database..........: Legacy DFU Version
----------Descriptors...............: None
```

Figure 14: GATT server - Open bootloader



```
Service--------------------------------------------------
--Service UUID......: FE59
--Service type......: 2800--Primary Service
--Service handle....: 0x9
--Name..............: FE59
--Name in database..: Secure DFU Service


------ Characteristic
----------UUID......................: 8ec90002-f315-4f60-9fb8-838830daea50
----------Type......................: 2803-Characteristic
----------Handle....................: 0xa
----------Value Handle..............: 0xb
----------Permissions...............: 0x4 : ['Write Without Response']
----------Name......................: 8ec90002-f315-4f60-9fb8-838830daea50
----------Name in database..........: DFU Packet
----------Descriptors...............: None


------ Characteristic
----------UUID......................: 8ec90001-f315-4f60-9fb8-838830daea50
----------Type......................: 2803-Characteristic
----------Handle....................: 0xc
----------Value Handle..............: 0xd
----------Permissions...............: 0x18 : ['Write', 'Notify']
----------Name......................: 8ec90001-f315-4f60-9fb8-838830daea50
----------Name in database..........: DFU Control Point
----------Descriptors...............: Yes
----------------Descriptor handle...: 0xe
----------------Descriptor value....: b'\x00\x00'
```

Figure 15: GATT server - Secure bootloader

5 Security analysis

In this section, we discuss the security of the DFU process:

- By discovering Services and Characteristics name and UUIDs, we may know which kind of bootloader that the board using: open or secure. See Section 4.5.

- By Remark 11, we see that the DFU process on open bootloader may have a high change of corrupt since it only checks the CRC after finishing the sending. While in secure bootloader, for large file, it sends the data by small parts of 4096 bytes, request checksum, execute, then sends next part,...
- By Section 4.3.1 and Remark 12, we see some problems in the validation of the firmware package of open bootloader:

  With a legitimate firmware packages, an attacker can modify the dat file and manifest.json file, modify the check-required values to the default values (see Section 3.2.1 to know where are the required values in the dat file so that the attacker can modify it correctly), then the validation will be skipped. The CRC algorithm is CRC-16-CCIT. The CRC checking algorithm is even optional and not implemented in some application.

  Hence, the attacker can craft a bad firmware package and pass the validation process.
- By Section 4.3.2, it seems difficult for an attacker to craft a bad firmware without the private key since the secure bootloader implements both CRC and signature validation.
- By Section 4.3.2, using the mechanism of validating firmware packages, if an attacker has access to a validated firmware package, the attacker can identify the version of hardware, softdevice that the target is using. The attacker does not need to do all the DFU process, just need to send the firmware dat file and see the response from the target. We can use the python application described in Section 4.4.

References

[1] Townsend, K., Cuf, C., Akiba, & Davidson, R. (2014). *Getting Started with Bluetooth Low Energy: Tools and Techniques for Low-Power Networking.* O'Reilly Media, Inc. https://www.oreilly.com/library/view/getting-started-with/9781491900550/

[2] Gupta, N. *Inside Bluetooth Low Energy.* (Artech House, 2016) https://us.artechhouse.com/Inside-Bluetooth-Low-Energy-Second-Edition-P1848.aspx

[3] Bluetooth Core Specification 4.0, available at https://www.bluetooth.com/specifications/specs/core-specification-4-0/

[4] Gitlab repository of the project: https://gitlab.eurecom.fr/lehh/spring2023-Security-analysis-of-Bluetooth-low-energy-Over-the-Air-firmware-updates

[5] Whad-client private project: https://github.com/whad-team/whad-client

[6] nRF Connect SDK: https://www.nordicsemi.com/Products/Development-software/nRF-Connect-SDK