Infection Risks of a Samsung Smart watch

Eurecom Semester Project

Professor:
Aurélien Francillon

Students:
Le Kim Hung   **Pham Duc Hien**

CONTRIBUTION IN THE REPORT

PART A – RISKS IN CONNECTION PROCEDURE

- Pham Duc Hien

PART B – RISK IN FIRMWARE AND GEARFIT'S APPLICATION

- Le Kim Hung

Table of Contents

PART A – RISKS IN CONNECTION PROCEDURE (Contributor - PHAM DUC HIEN)

Introductions

We have been interested in the risks of connection between Samsung smart watch and other devices. By designed, the smart watch can only connect to deviceswhich use a Samsung application named Gear Fit Manager. We wanted to know if a device would be able to connect to the watch without the vendor-specific application.

Because the only way to connect with the smartwatch is to use Bluetooth, we decided to research on this wireless technology. As Samsung smart watch is introduced to use Bluetooth v4.0 (also known as Bluetooth Smart or Bluetooth Low Energy), which has been known to be less secure than classical Bluetooth version in some extents, we hoped to find potential risks of the connection of the smart watch by studying Bluetooth v4.0 core specification and related scientific papers.

Methods

We tried 3 different approaches: HCI logs, reversed apk file, Ubertooth One in order to study the feasibility of attacking the connection of the Samsung smart watch. During the report, our comments and conclusions were made at the end of each approach or, sometimes,each sub-section of an approach.

Below are some tools we used in each approach:

- HCI logs: Python Bluetooth libraries (PyBluez[10]),Wireshark, HCI logs
- Reversed apk file:Java decompiler(jdk-gui), adb logs from Android phones
- Ubertooth One:Ubertooth One device([4] and [5]), Wireshark

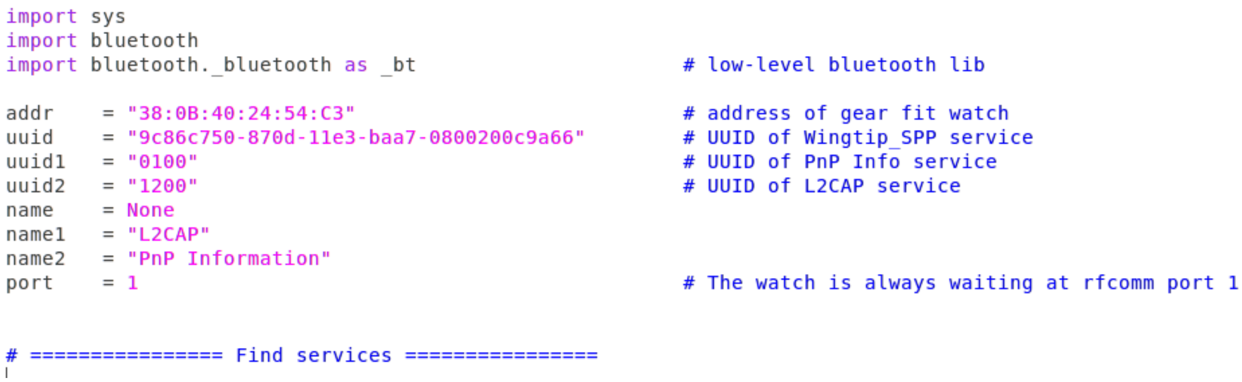
The reason why we tried these 3 approaches was that our knowledge in Bluetooth v4.0 has generally grown up from zero as we studied this new Bluetooth version during the course of our project. The more we understood this technology (and also the communication protocol of the smart watch), the larger numbers of approaches we could come up with.

1. HCI Logs Approach

This is an application-level approach, in which we tried to send to the watch as many packets used by Gear Fit Manager as possible. These packets were observed during connection process of the smart watch and another device.

Firstly, we investigated the HCI logs of connection process between a phone and the smartwatch. HCI logs were obtained by enabling an option in Android Kit Kat phones. Secondly, we wrote a Python script to apply our knowledge from the HCI logs. Our goal when working on the HCI logs was to create a connection between a device and the smart watch without using Gear Fit Manager. Let's see how far we reached in this approach.

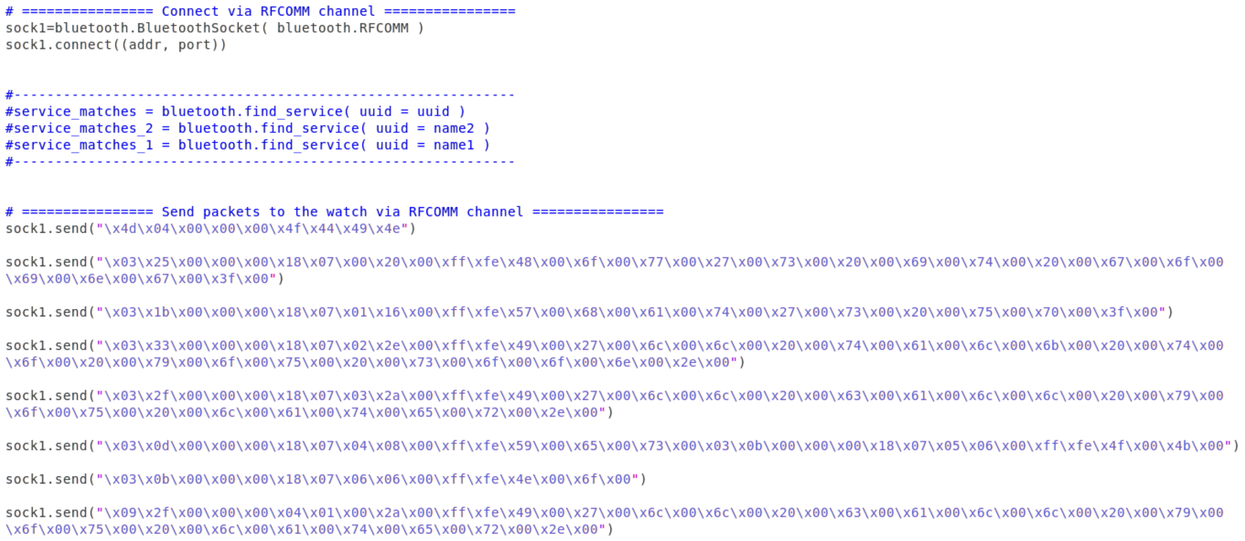
From the HCI logs, we found that a RFCOMM channel was set up for data exchange. More importantly, a specific service (named WingTip) in the watch must have been accessed by the phone before establishing the RFCOMM channel. The special value which must be used to access that service is: "9c86c750-870d-11e3-baa7-0800200c9a66". So we put it in our Python script:



```
import sys
import bluetooth
import bluetooth._bluetooth as _bt                       # low-level bluetooth lib

addr    = "38:0B:40:24:54:C3"                            # address of gear fit watch
uuid    = "9c86c750-870d-11e3-baa7-0800200c9a66"         # UUID of Wingtip_SPP service
uuid1   = "0100"                                         # UUID of PnP Info service
uuid2   = "1200"                                         # UUID of L2CAP service
name    = None
name1   = "L2CAP"
name2   = "PnP Information"
port    = 1                                              # The watch is always waiting at rfcomm port 1


# ================ Find services ================
```

Special value used to access WingTip service on the watch

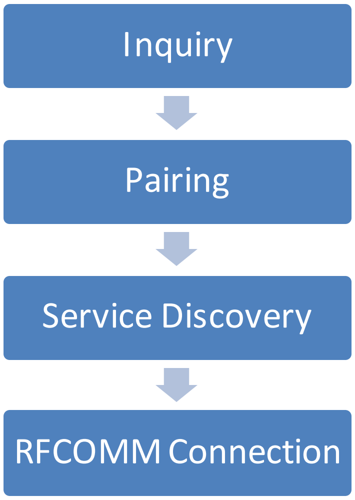
In RFCOMM channel, lots of information were sent until the connection process ended and we captured them via HCI logs. We thought they could be specific values, which must be sent from a device to the smart watch for every connection attempts. Therefore we included them into our Python script as well:



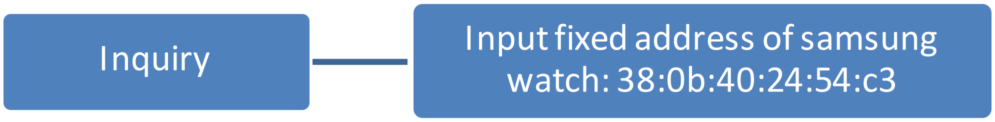
```
# ================ Connect via RFCOMM channel ================
sock1=bluetooth.BluetoothSocket( bluetooth.RFCOMM )
sock1.connect((addr, port))


#------------------------------------------------------------
#service_matches = bluetooth.find_service( uuid = uuid )
#service_matches_2 = bluetooth.find_service( uuid = name2 )
#service_matches_1 = bluetooth.find_service( uuid = name1 )
#------------------------------------------------------------


# ================ Send packets to the watch via RFCOMM channel ================
sock1.send("\x4d\x04\x00\x00\x00\x4f\x44\x49\x4e")

sock1.send("\x03\x25\x00\x00\x00\x18\x07\x00\x20\x00\xff\xfe\x48\x00\x6f\x00\x77\x00\x27\x00\x73\x00\x20\x00\x69\x00\x74\x00\x20\x00\x67\x00\x6f\x00
\x69\x00\x6e\x00\x67\x00\x3f\x00")

sock1.send("\x03\x1b\x00\x00\x00\x18\x07\x01\x16\x00\xff\xfe\x57\x00\x68\x00\x61\x00\x74\x00\x27\x00\x73\x00\x20\x00\x75\x00\x70\x00\x3f\x00")

sock1.send("\x03\x33\x00\x00\x00\x18\x07\x02\x2e\x00\xff\xfe\x49\x00\x27\x00\x6c\x00\x6c\x00\x20\x00\x74\x00\x61\x00\x6c\x00\x6b\x00\x20\x00\x74\x00
\x6f\x00\x20\x00\x79\x00\x6f\x00\x75\x00\x20\x00\x73\x00\x6f\x00\x6f\x00\x6e\x00\x2e\x00")

sock1.send("\x03\x2f\x00\x00\x00\x18\x07\x03\x2a\x00\xff\xfe\x49\x00\x27\x00\x6c\x00\x6c\x00\x20\x00\x63\x00\x61\x00\x6c\x00\x6c\x00\x20\x00\x79\x00
\x6f\x00\x75\x00\x20\x00\x6c\x00\x61\x00\x74\x00\x65\x00\x72\x00\x2e\x00")

sock1.send("\x03\x0d\x00\x00\x00\x18\x07\x04\x08\x00\xff\xfe\x59\x00\x65\x00\x73\x00\x03\x0b\x00\x00\x00\x18\x07\x05\x06\x00\xff\xfe\x4f\x00\x4b\x00")

sock1.send("\x03\x0b\x00\x00\x00\x18\x07\x06\x06\x00\xff\xfe\x4e\x00\x6f\x00")

sock1.send("\x09\x2f\x00\x00\x00\x04\x01\x00\x2a\x00\xff\xfe\x49\x00\x27\x00\x6c\x00\x6c\x00\x20\x00\x63\x00\x61\x00\x6c\x00\x6c\x00\x20\x00\x79\x00
\x6f\x00\x75\x00\x20\x00\x6c\x00\x61\x00\x74\x00\x65\x00\x72\x00\x2e\x00")
```

Values observed in HCI logs

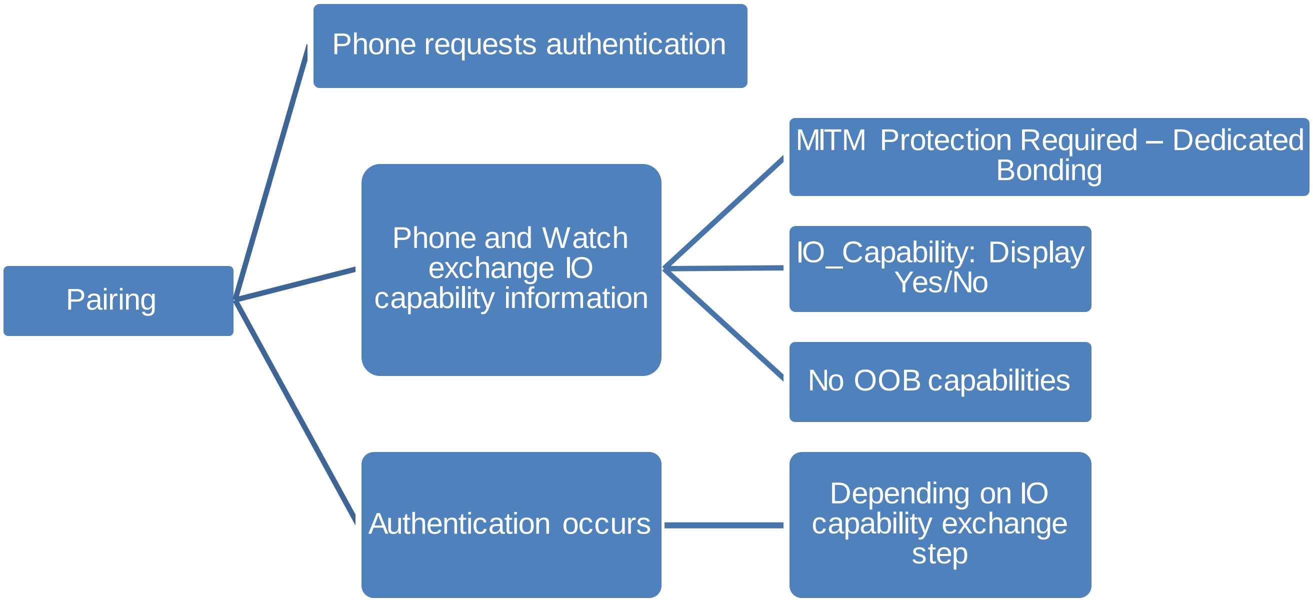
We sketched a schema of the connection process based on HCI logs.In our next approach, we also sketched another schema of connection procedure basing on the reversed apk file of Gear Fit Manager. We hoped that by looking at both of them, we would be able to identify what had probably been missing. Below is the schema of connection procedure based on HCI logs:



1.1. Inquiry



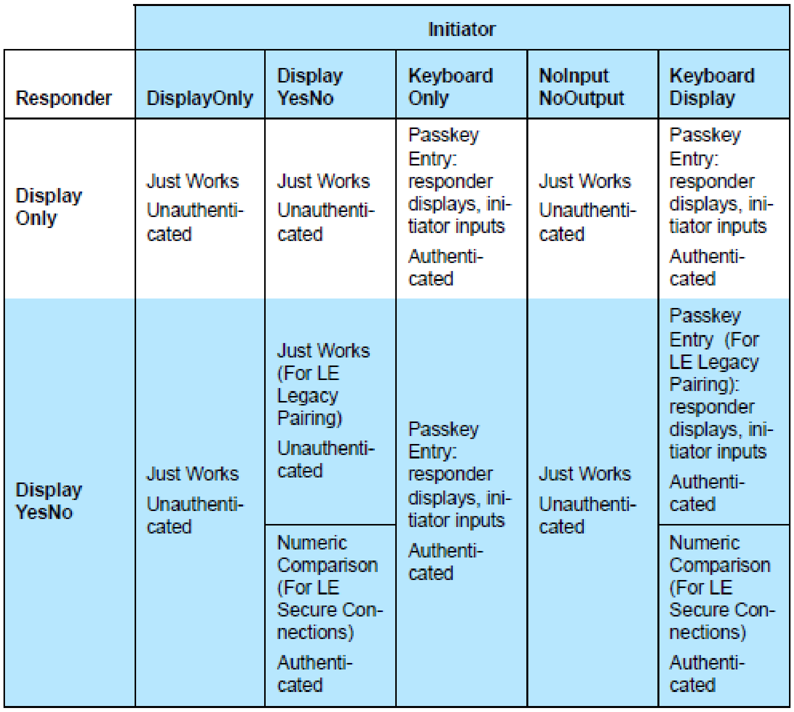
1.2. Pairing (Authentication Type)


Pairing
Phone requests authentication
Phone and Watch exchange IO capability information
MITM Protection Required – Dedicated Bonding
IO_Capability: Display Yes/No
No OOB capabilities
Authentication occurs
Depending on IO capability exchange step

From values collected during this step, we found a way to understand the authentication type of the connection.

From HCI logs, we knew that Samsung smart watch chose “MITM Protection Required – Dedicated Bonding” option. Consequently, its authentication type was determined based on the IO_Capability parameters.The smart watch and the phone first exchanged their IO capability then based on the information to decide their authentication.

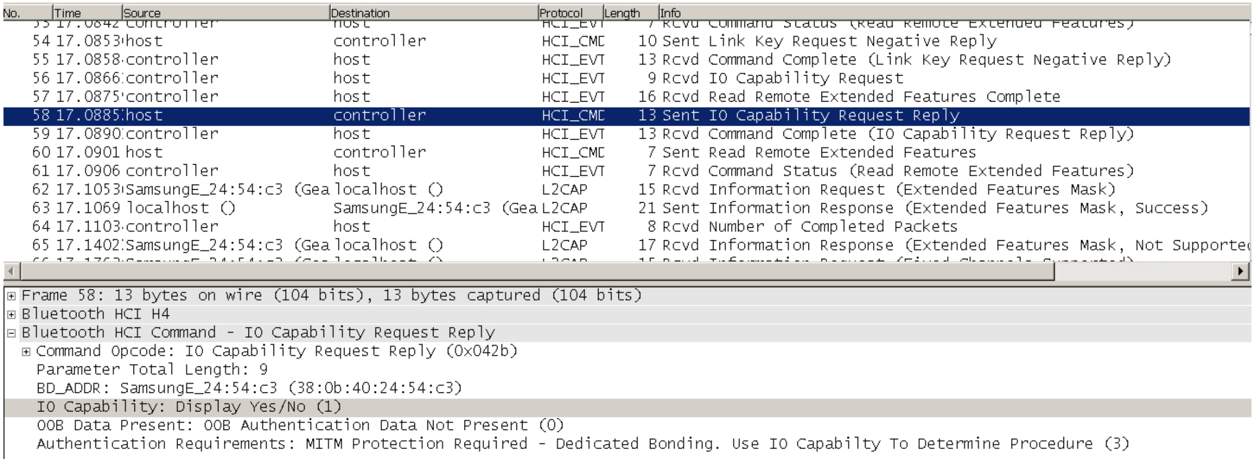
We foundthe below Authentication table, which introduced many cases of determining an authentication type of a Bluetooth Low Energy communication.



	Initiator				
Responder	DisplayOnly	Display YesNo	Keyboard Only	NoInput NoOutput	Keyboard Display
Display Only	Just Works Unauthenticated	Just Works Unauthenticated	Passkey Entry: responder displays, initiator inputs Authenticated	Just Works Unauthenticated	Passkey Entry: responder displays, initiator inputs Authenticated
Display YesNo	Just Works Unauthenticated	Just Works (For LE Legacy Pairing) Unauthenticated Numeric Comparison (For LE Secure Connections) Authenticated	Passkey Entry: responder displays, initiator inputs Authenticated	Just Works Unauthenticated	Passkey Entry (For LE Legacy Pairing): responder displays, initiator inputs Authenticated Numeric Comparison (For LE Secure Connections) Authenticated

Authentication Table – Figure taken from [2]

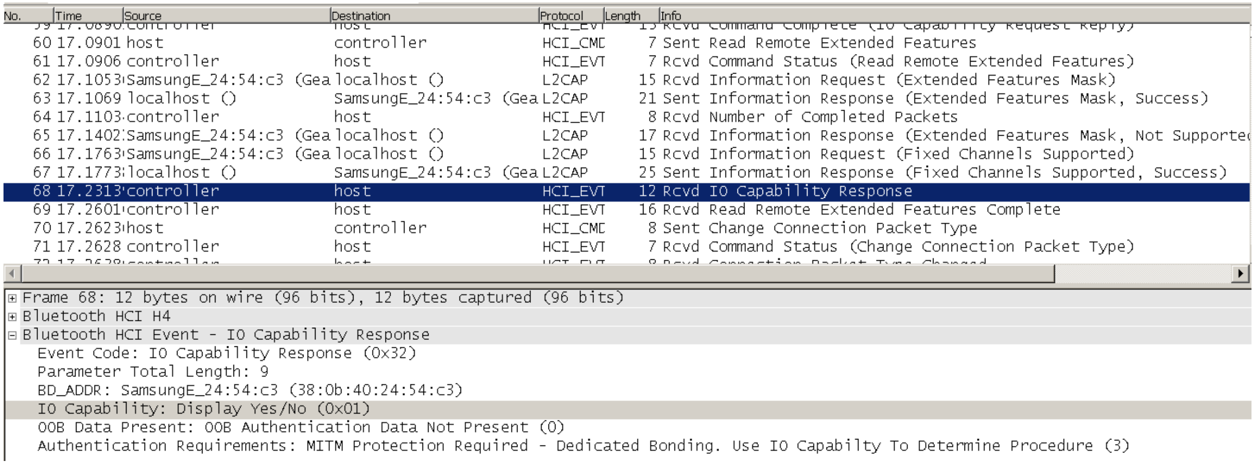
We then compared the table with what we had in our captured HCI logs to understand which type of authentication was made by the smart watch and the phone. Below is the I/O Capability information sent by the phone:



```
No.  Time     Source                  Destination             Protocol  Length Info
  53 17.0842 controller               host                    HCI_EVT      7 Rcvd Command Status (Read Remote Extended Features)
  54 17.0853 host                     controller              HCI_CMD     10 Sent Link Key Request Negative Reply
  55 17.0858 controller               host                    HCI_EVT     13 Rcvd Command Complete (Link Key Request Negative Reply)
  56 17.0866 controller               host                    HCI_EVT      9 Rcvd IO Capability Request
  57 17.0875 controller               host                    HCI_EVT     16 Rcvd Read Remote Extended Features Complete
  58 17.0885 host                     controller              HCI_CMD     13 Sent IO Capability Request Reply
  59 17.0890 controller               host                    HCI_EVT     13 Rcvd Command Complete (IO Capability Request Reply)
  60 17.0901 host                     controller              HCI_CMD      7 Sent Read Remote Extended Features
  61 17.0906 controller               host                    HCI_EVT      7 Rcvd Command Status (Read Remote Extended Features)
  62 17.1053 SamsungE_24:54:c3 (Gea   localhost ()            L2CAP       15 Rcvd Information Request (Extended Features Mask)
  63 17.1069 localhost ()             SamsungE_24:54:c3 (Gea  L2CAP       21 Sent Information Response (Extended Features Mask, Success)
  64 17.1103 controller               host                    HCI_EVT      8 Rcvd Number of Completed Packets
  65 17.1402 SamsungE_24:54:c3 (Gea   localhost ()            L2CAP       17 Rcvd Information Response (Extended Features Mask, Not Supportec

⊞ Frame 58: 13 bytes on wire (104 bits), 13 bytes captured (104 bits)
⊞ Bluetooth HCI H4
⊟ Bluetooth HCI Command - IO Capability Request Reply
  ⊞ Command Opcode: IO Capability Request Reply (0x042b)
    Parameter Total Length: 9
    BD_ADDR: SamsungE_24:54:c3 (38:0b:40:24:54:c3)
    IO Capability: Display Yes/No (1)
    OOB Data Present: OOB Authentication Data Not Present (0)
    Authentication Requirements: MITM Protection Required - Dedicated Bonding. Use IO Capabilty To Determine Procedure (3)
```

I/O Capability information of the phone

And the I/O Capability information of the smart watch:



```
No.  Time     Source                  Destination             Protocol Length Info
60 17.0901 host                  controller              HCI_CMD   7 Sent Read Remote Extended Features
61 17.0906 controller            host                    HCI_EVT   7 Rcvd Command Status (Read Remote Extended Features)
62 17.1053 SamsungE_24:54:c3 (Gea localhost ()            L2CAP    15 Rcvd Information Request (Extended Features Mask)
63 17.1069 localhost ()          SamsungE_24:54:c3 (Gea  L2CAP    21 Sent Information Response (Extended Features Mask, Success)
64 17.1103 controller            host                    HCI_EVT   8 Rcvd Number of Completed Packets
65 17.1402 SamsungE_24:54:c3 (Gea localhost ()            L2CAP    17 Rcvd Information Response (Extended Features Mask, Not Supporte
66 17.1763 SamsungE_24:54:c3 (Gea localhost ()            L2CAP    15 Rcvd Information Request (Fixed Channels Supported)
67 17.1773 localhost ()          SamsungE_24:54:c3 (Gea  L2CAP    25 Sent Information Response (Fixed Channels Supported, Success)
68 17.2313 controller            host                    HCI_EVT  12 Rcvd IO Capability Response
69 17.2601 controller            host                    HCI_EVT  16 Rcvd Read Remote Extended Features Complete
70 17.2623 host                  controller              HCI_CMD   8 Sent Change Connection Packet Type
71 17.2628 controller            host                    HCI_EVT   7 Rcvd Command Status (Change Connection Packet Type)

⊞ Frame 68: 12 bytes on wire (96 bits), 12 bytes captured (96 bits)
⊞ Bluetooth HCI H4
⊟ Bluetooth HCI Event - IO Capability Response
    Event Code: IO Capability Response (0x32)
    Parameter Total Length: 9
    BD_ADDR: SamsungE_24:54:c3 (38:0b:40:24:54:c3)
    IO Capability: Display Yes/No (0x01)
    OOB Data Present: OOB Authentication Data Not Present (0)
    Authentication Requirements: MITM Protection Required - Dedicated Bonding. Use IO Capabilty To Determine Procedure (3)
```

I/O Capability information of the watch

1.2.1. Conclusions

Because the two devices chose their I/O capability as “Display Yes/No” and they actually exchanged a numeric value to each other for verifying during authentication procedure, we realized the authentication between the smart watch and the phone wouldprobably be Numeric Comparison.

Most importantly, the authentication type was not Out-of-band (OOB), which would be the most difficult type for potential exploitations. This gave us an idea to use Ubertooth One to sniff and bypass the encryption process, which is mentioned in latter parts of this report.

1.3. Service Discovery

An initiating device searches for WingTip service on the smart watch. Specific value “9c86c750-870d-11e3-baa7-0800200c9a66” was used to access this service. This value is fixed by Samsung and must be used if one wants to connect to the smart watch.

1.4. RFCOMM Connection

A bunch of packets was sent to the smart watch. We wrote a Python script which would send every of these packets from my personal computer to the smart watch.

1.5. Results

After combining observed values together, we found our Python script still unable to create a connection between a device (in this case a personal laptop) and the smart watch.

1.6. Comments and Conclusions

- We understood the authentication type of the connection between the smart watch and a phone.

- Although we used all RFCOMM values which were visible in HCI logs, a connection could not be established. The smart watch kept waiting for a signal from Gear Fit Manager. This could be understood that, by using knowledge from HCI logs, we had not sent enough information as the smart watch expected.It meant using only HCI logs was not sufficient to attack the communication of Samsung smart watch, and that we might need another method to sniff packets exchanged over air. But first, we wanted to know which kind of information was missing. Let us move into our second approach "Reversed apk file".

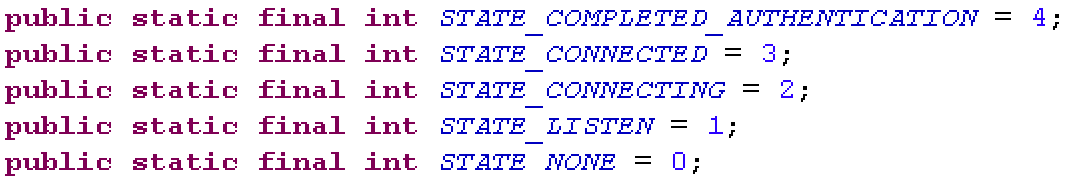
2. Reversed APK File Approach

In this approach, our first step was to collect an adb log of a phone, which was using Gear Fit Manager, during its connectionprocess to the smart watch. Our second step was to look into the reversed apk file of Gear Fit Manager, compare it with the adb logs and build a schema of connection process. Finally, we compared the new schema with the one in HCI logs approach

By this way, we believed that we would be able to find missing information when we used HCI logs approach.

2.1. States of Connections

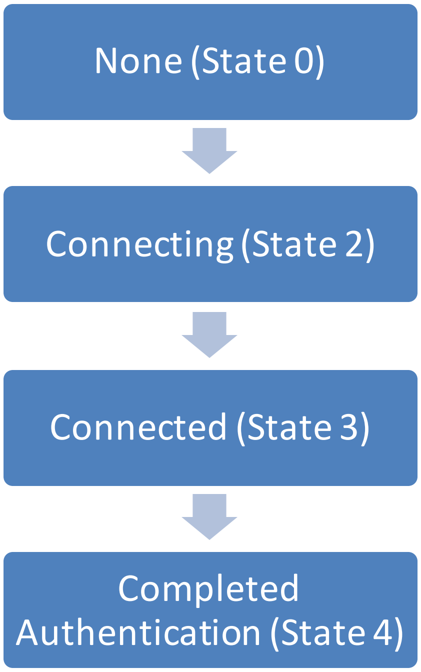
From reversed apk file of Gear Fit Manager, we knew how it would define states of its connection with the smart watch like below:



```
public static final int STATE_COMPLETED_AUTHENTICATION = 4;
public static final int STATE_CONNECTED = 3;
public static final int STATE_CONNECTING = 2;
public static final int STATE_LISTEN = 1;
public static final int STATE_NONE = 0;
```

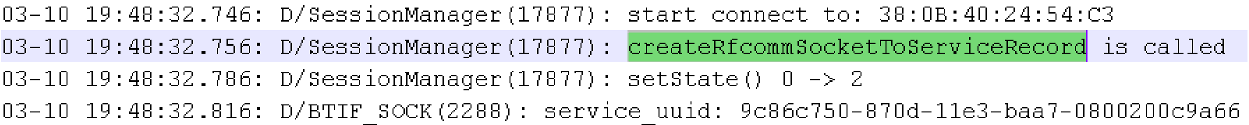
States of Connections

2.2. Schema



2.2.1. Connecting(changing from state 0 to state 2)

This is what we captured from the adb logs during this step:

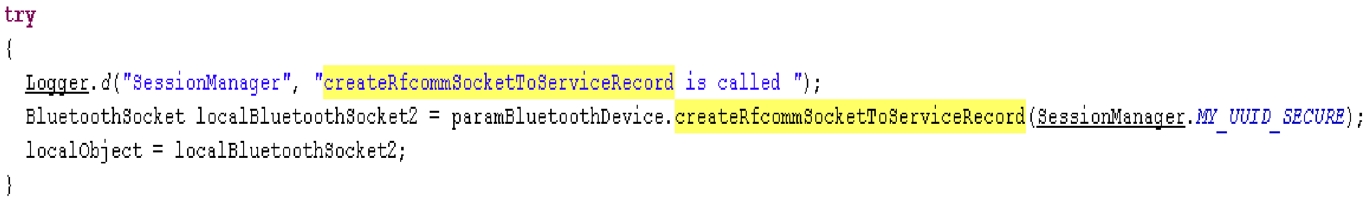


```
03-10 19:48:32.746: D/SessionManager(17877): start connect to: 38:0B:40:24:54:C3
03-10 19:48:32.756: D/SessionManager(17877): createRfcommSocketToServiceRecord is called
03-10 19:48:32.786: D/SessionManager(17877): setState() 0 -> 2
03-10 19:48:32.816: D/BTIF_SOCK(2288): service_uuid: 9c86c750-870d-11e3-baa7-0800200c9a66
```

Adb logs - Connecting state

We noted that 38:0B:40:24:54:C3 was the address of the smart watch and “9c86c750-870d-11e3-baa7-0800200c9a66” was the specific value used to access WingTip service in the smart watch. Let us look into the reversed apk file of Gear Fit Manager to understand what really happened.

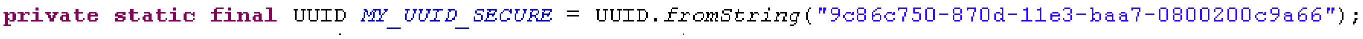
Below is how Gear Fit Manager accessed WingTip service on the smart watch:



```
try
{
  Logger.d("SessionManager", "createRfcommSocketToServiceRecord is called ");
  BluetoothSocket localBluetoothSocket2 = paramBluetoothDevice.createRfcommSocketToServiceRecord(SessionManager.MY_UUID_SECURE);
  localObject = localBluetoothSocket2;
}
```

Reversed apk file of Gear Fit Manager

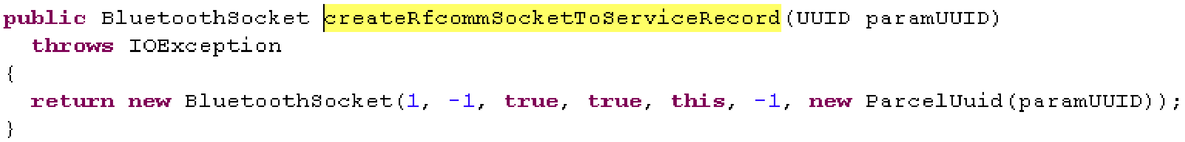
, where MY_UUID_SECURE was also the specific value we found in HCI logs approach:



```
private static final UUID MY_UUID_SECURE = UUID.fromString("9c86c750-870d-11e3-baa7-0800200c9a66");
```

Special value used to access WingTip service on the watch

A closer look on “createRfcommSocketToServiceRecord” method:



```
public BluetoothSocket createRfcommSocketToServiceRecord(UUID paramUUID)
  throws IOException
{
  return new BluetoothSocket(1, -1, true, true, this, -1, new ParcelUuid(paramUUID));
}
```

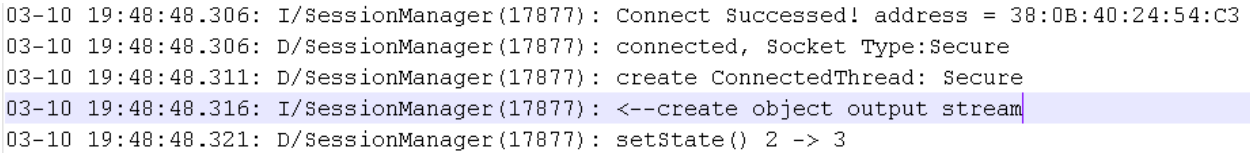
Reversed apk file of Gear Fit Manager

2.2.1.1. Comments

So far, this step seemed similar to what we had known in HCI logs. There had been no new values shown to us. Let us move into next step of the schema.

2.2.2. **Connected**(changing from state 2 to state 3)

This is what we captured from the adb logs during this step:



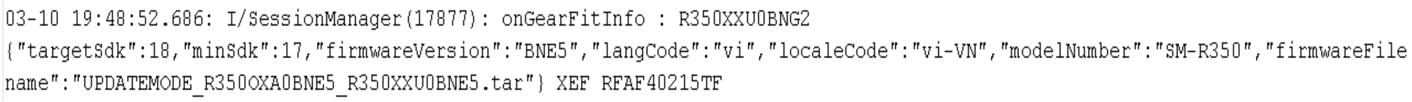
```
03-10 19:48:48.306: I/SessionManager(17877): Connect Successed! address = 38:0B:40:24:54:C3
03-10 19:48:48.306: D/SessionManager(17877): connected, Socket Type:Secure
03-10 19:48:48.311: D/SessionManager(17877): create ConnectedThread: Secure
03-10 19:48:48.316: I/SessionManager(17877): <--create object output stream
03-10 19:48:48.321: D/SessionManager(17877): setState() 2 -> 3
```

Adb logs - Connected state

In this step, looking into the reversed apk did not give us many ideas about what really happened. But basing on the information from adb logs, we could understand that an IO capabilities exchanging was happening, which could then lead to a decision on authentication type of the connection.

2.2.3. **Completed Authentication**(changing from state 3 to state 4)

From adb log, we knew that Gear Fit Manager createda packet like below then sent it to the watch:



```
03-10 19:48:52.686: I/SessionManager(17877): onGearFitInfo : R350XXU0BNG2
{"targetSdk":18,"minSdk":17,"firmwareVersion":"BNE5","langCode":"vi","localeCode":"vi-VN","modelNumber":"SM-R350","firmwareFile
name":"UPDATEMODE_R350OXA0BNE5_R350XXU0BNE5.tar"} XEF RFAF40215TF
```

Information which had not been observed in HCI logs

We tried to look into the reversed apk file and traced back to understand how the packet had been created. But all functions related to this packet had not been reversed successfully by our tool jd-gui. It was probably because Samsung considered those functions and this packet as secrets and did not want to expose them.

2.2.3.1. **Comments**

- We did not observe that packet through our previous HCI logs approach. Therefore, it would be the information which our previous approach had been lack of.

- Exploitations into the reversed apk file to find the hidden functions would be a potential future research. In this project, we chose another way which was to use Bluetooth sniffing techniques.

2.3. Conclusions

Up to this point, we knew which information we missed in our first approach. We also knew the authentication type of the connection between the smart watch and a phone. Therefore, we decided to make a third approach which employed Ubertooth One to sniff the missing information via Bluetooth communication.

3. Ubertooth OneApproach

The target of this approach was to collect missing packets from two previous approaches. Therefore, unlike 2 above approaches, we worked with link layer controller and air interface of Bluetooth Low Energy in this approach. We planned to use Ubertooth One as a Bluetooth sniffing device. There were 2 main reasons why we chose this device.

Firstly, the device supported the ability of sniffing Bluetooth Low Energy which was supposed to be used by Samsung smart watch.

Secondly, there was a scientific paper [6] which introduced sniffing and encryption by-passing techniques for Bluetooth Low Energy, by using this device.

Known drawbacks of the approach

Actually, if targeted devices used Out-of-band method for their authentication and exchanging keys,then the techniques introduced by the scientific paperwould not work. But in HCI logs approach, we already found that Samsung smart watch did not used Out-of-band method. Therefore, this drawback of the techniques did not affect our Ubertooth One approach.

The authors of Ubertooth project on GitHub published a software to be used with

Ubertooth One device. Their software was supposed to capture Bluetooth Low Energy packets. But when using that software, we did not capture enough packets for pairing process of 2 devices. So we needed to do a workaround, looked into its source code to see what happened and posted questions to the support team of Ubertooth project.

Although we did not reach the goal of this approach, we still found the reason behind, and some other interesting facts which would be useful for future research.

3.1. Sniffing

3.1.1. Hopping Frequency

When using Bluetooth Low Energy, devices do not stay in the same frequency for transferring data. They actually hop between frequency channels and stay in each channel for a short time to transfer data then jump to other channels. This means if one wants to sniff these data, he has to move between these channels also.

Therefore, before using Ubertooth One to sniff the communication between the smart watch and a phone, we needed to understand the hopping formula. From now on let us call Bluetooth Low Energy as BTLE.

BTLE splits the 2.4 GHz ISM band into 40 channels (to be more exact, from 2402 to 2480 GHz)

- 3 Advertising Channels
- 37 Data Channels
- fn = 2402 + 2n MHz

(The numbers taken from [13])

An initiating device, in our case the phone, would decide which channel to hop into first, and the increment for later channel hopping. All these information was captured by Ubertooth and would be introduced in next sections.

3.1.2. Advertising Channels:

BTLE uses exactly 3 advertising channels. It is very important to know this, as Ubertooth Onedevice can only sit on 1 channel at a time. These 3 channels are:

- 2402, 2426, 2480 MHz, indexed as channel 37, 38 and 39

These channels are used by peripherals to advertise their presence. In our case, the smart watch played the role of a peripheral and kept broadcasting its advertising packets to anybody scanning.

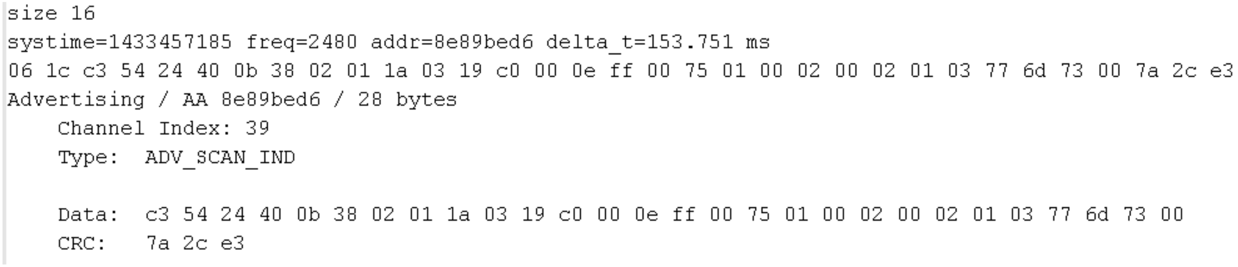
Let us analyze the advertising packets captured by Ubertooth One.There are many types of Advertising PDU like below:

PDU	Type
0000	ADV_IND
0001	ADV_DIRECT_IND
0010	ADV_NONCONN_IND
0011	SCAN_REQ
0100	SCAN_RSP
0101	CONNECT_REQ
0110	ADV_SCAN_IND

(The details taken from [1])

3.1.2.1. **Results**

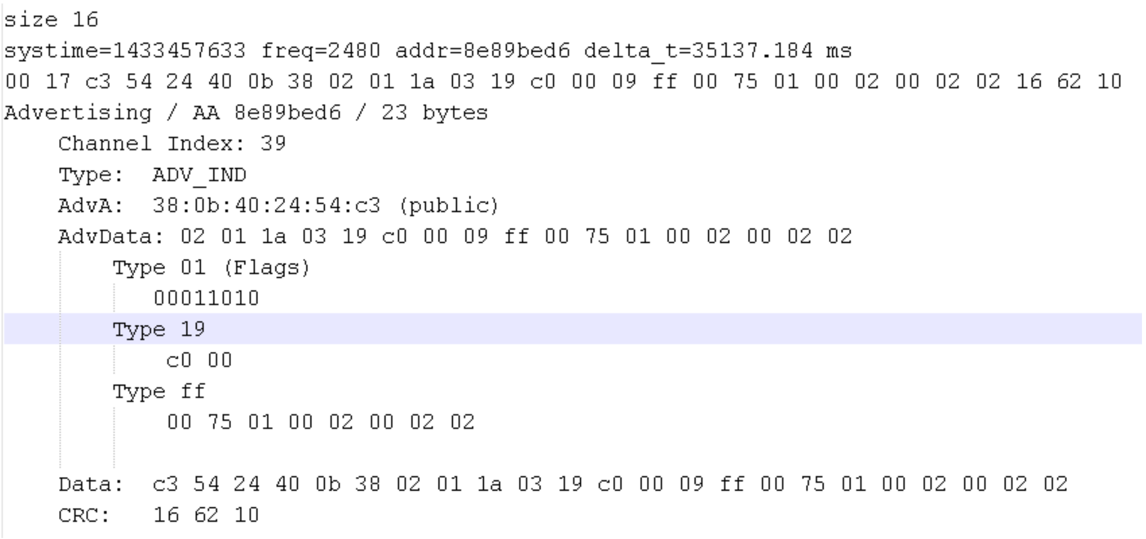
But most often, we captured two types of them: ADV_IND and ADV_SCAN_IND, which respectively are Advertising Indirectly and Advertising Scannable Indirectly. ADV_SCAN_IND (06) packets arrived before doing any pairing activities, like in the figure below:



```
size 16
systime=1433457185 freq=2480 addr=8e89bed6 delta_t=153.751 ms
06 1c c3 54 24 40 0b 38 02 01 1a 03 19 c0 00 0e ff 00 75 01 00 02 00 02 01 03 77 6d 73 00 7a 2c e3
Advertising / AA 8e89bed6 / 28 bytes
    Channel Index: 39
    Type:  ADV_SCAN_IND

    Data:  c3 54 24 40 0b 38 02 01 1a 03 19 c0 00 0e ff 00 75 01 00 02 00 02 01 03 77 6d 73 00
    CRC:   7a 2c e3
```

Screenshot of ADV_SCAN_IND (06) packets

ADV_SCAN_IND(00) packets were captured after pairing the smart watch with a phone, like in the figure below:

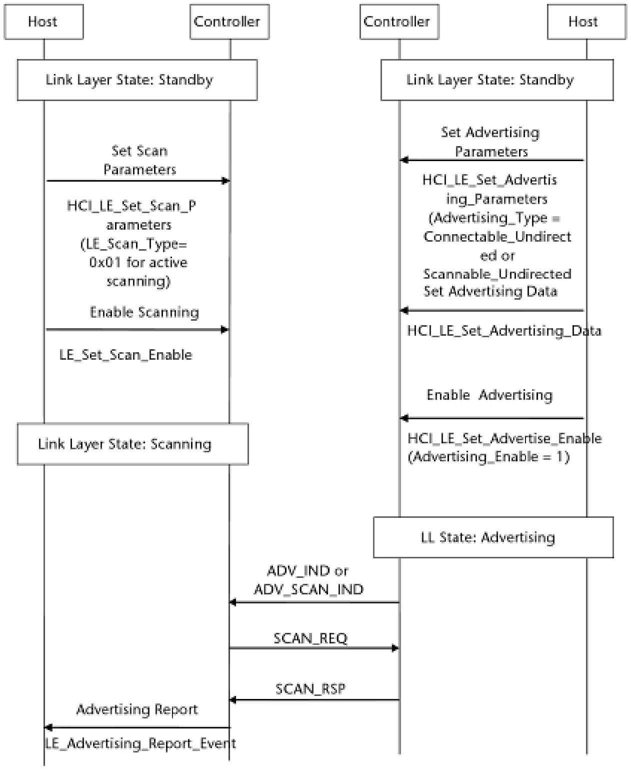


```
size 16
systime=1433457633 freq=2480 addr=8e89bed6 delta_t=35137.184 ms
00 17 c3 54 24 40 0b 38 02 01 1a 03 19 c0 00 09 ff 00 75 01 00 02 00 02 02 16 62 10
Advertising / AA 8e89bed6 / 23 bytes
    Channel Index: 39
    Type:  ADV_IND
    AdvA:  38:0b:40:24:54:c3 (public)
    AdvData: 02 01 1a 03 19 c0 00 09 ff 00 75 01 00 02 00 02 02
        Type 01 (Flags)
           00011010
        Type 19
            c0 00
        Type ff
            00 75 01 00 02 00 02 02

    Data:  c3 54 24 40 0b 38 02 01 1a 03 19 c0 00 09 ff 00 75 01 00 02 00 02 02
    CRC:   16 62 10
```

Screenshot of ADV_IND (00) packets

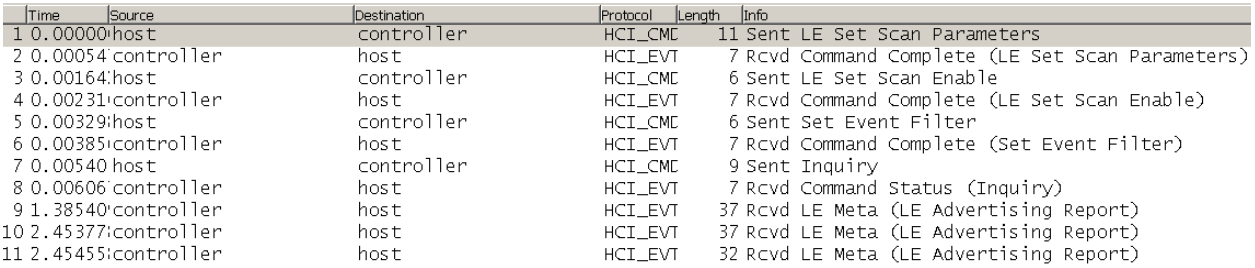
3.1.2.2. Comments

To understand this behavior, we looked into the schema of BTLE advertising/scanning process.


Host
Controller
Controller
Host
Link Layer State: Standby
Link Layer State: Standby
Set Scan Parameters
HCI_LE_Set_Scan_Parameters (LE_Scan_Type= 0x01 for active scanning)
Enable Scanning
LE_Set_Scan_Enable
Link Layer State: Scanning
Set Advertising Parameters
HCI_LE_Set_Advertising_Parameters (Advertising_Type = Connectable_Undirected or Scannable_Undirected
Set Advertising Data
HCI_LE_Set_Advertising_Data
Enable Advertising
HCI_LE_Set_Advertise_Enable (Advertising_Enable = 1)
LL State: Advertising
ADV_IND or ADV_SCAN_IND
SCAN_REQ
SCAN_RSP
Advertising Report
LE_Advertising_Report_Event

Active Scanning - Figure taken from[9]

Let us combine packets captured by Ubertooth Oneto an HCI log we had captured before:



	Time	Source	Destination	Protocol	Length	Info
1	0.00000	host	controller	HCI_CMD	11	Sent LE Set Scan Parameters
2	0.00054	controller	host	HCI_EVT	7	Rcvd Command Complete (LE Set Scan Parameters)
3	0.00164	host	controller	HCI_CMD	6	Sent LE Set Scan Enable
4	0.00231	controller	host	HCI_EVT	7	Rcvd Command Complete (LE Set Scan Enable)
5	0.00329	host	controller	HCI_CMD	6	Sent Set Event Filter
6	0.00385	controller	host	HCI_EVT	7	Rcvd Command Complete (Set Event Filter)
7	0.00540	host	controller	HCI_CMD	9	Sent Inquiry
8	0.00606	controller	host	HCI_EVT	7	Rcvd Command Status (Inquiry)
9	1.38540	controller	host	HCI_EVT	37	Rcvd LE Meta (LE Advertising Report)
10	2.45377	controller	host	HCI_EVT	37	Rcvd LE Meta (LE Advertising Report)
11	2.45455	controller	host	HCI_EVT	32	Rcvd LE Meta (LE Advertising Report)

LE_Set_Scan_Parameters and LE_Set_Scan_Enable commands in HCI logs

Now we could envision the flow of packets between the smart watch and a phone. LE_Set_Scan_Parameters andLE_Set_Scan_Enable commandswere sent first, then ADV_IND or ADV_SCAN_IND. Here we could appreciate an advantage of Ubertooth One that is to capture packets sent at link layer and not visible in HCI logs (2 packets ADV_IND and ADV_SCAN_IND).

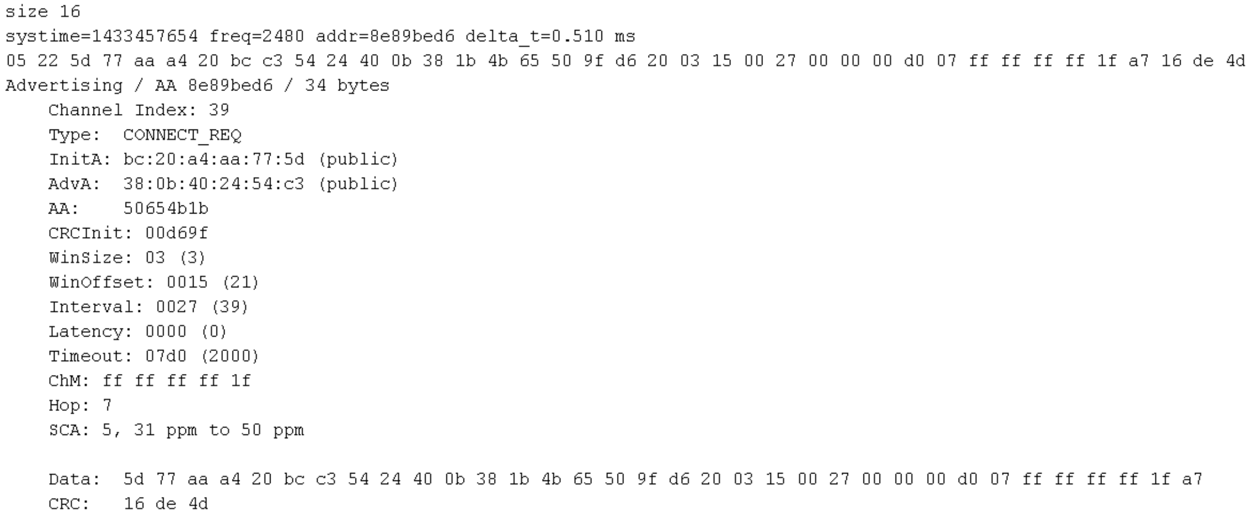
3.1.2.3. Conclusions

Finally, we understood the observed behavior of the smart watch.Before any pairing, the smart watch would send ADV_SCAN_IND packet as it wanted to be visible to other devices. In another hand, after being paired, it would hide itselffrom the world and therefore send ADV_IND. Note that at this time, on the phone side, Gear Fit Manager already saved the smart watch information (like BD_ADDR, device name, etc.) and did not needscanning activities to detect the watch.

3.1.3. Connection Request

After sniffing the advertising packets of the smart watch, we tried to obtain CONNECT_REQ packet which would be sent to the smart watch.In order to establish a connection, an initiating device must send this packet to a peripheral device (the smart watch). The packet is very important as it defines which channel to start from and also the hopping increment. A peripheral device (the smart watch) requires this packet to hop along with its initiating device (the phone) and therefore, we need this packet to follow them.

Below is the CONNECT_REQ packet we captured:



```
size 16
systime=1433457654 freq=2480 addr=8e89bed6 delta_t=0.510 ms
05 22 5d 77 aa a4 20 bc c3 54 24 40 0b 38 1b 4b 65 50 9f d6 20 03 15 00 27 00 00 00 d0 07 ff ff ff ff 1f a7 16 de 4d
Advertising / AA 8e89bed6 / 34 bytes
    Channel Index: 39
    Type:  CONNECT_REQ
    InitA: bc:20:a4:aa:77:5d (public)
    AdvA:  38:0b:40:24:54:c3 (public)
    AA:    50654b1b
    CRCInit: 00d69f
    WinSize: 03 (3)
    WinOffset: 0015 (21)
    Interval: 0027 (39)
    Latency: 0000 (0)
    Timeout: 07d0 (2000)
    ChM: ff ff ff ff 1f
    Hop: 7
    SCA: 5, 31 ppm to 50 ppm

    Data:  5d 77 aa a4 20 bc c3 54 24 40 0b 38 1b 4b 65 50 9f d6 20 03 15 00 27 00 00 00 d0 07 ff ff ff ff 1f a7
    CRC:   16 de 4d
```

Screenshot of CONNECT_REQ (05) packets

Interesting information in this packet could be the starting channel which is “Hop: 7” in the figure. It was also the hop increment used for later channel hopping, i.e next n channels would be (7 + 7n) mod 37.

3.1.3.1. Conclusions

We found this step quite trivial because we could occasionally capture the packet. We posted a question to the support team of Ubertooth One to solve this problem. Then we understood that the chance of capturing this packet was only $\frac{1}{3}$. The reason was that Ubertooth One device could only sit on 1 advertising channel at a time while the smart watch and the phone could send their CONN_REQ at any of 3 advertising channels (37, 38 and 39).

3.2. Key Distribution

After collecting CONN_REQ packet, we would be able to capture data packets sent over data channels. These channels and the hopping increment were defined in the CONN_REQ by the initiating device (the phone). Data packets exchanged over data channels were encrypted using a session key. This session key was created based on a Long-Term Key which had been mutually decided by the smart watch and the phone. We found a solution to obtain this Long-Term Key, which is introduced in the next section “Potential Risks”.At this point, we needed theoretical knowledge in key distribution of BTLE.

From BTLE documents [3], we knew that there were three types of keys needed for the data encryption.

- Long-Term Key (LTK) is used to create a Session Key for each Link Layer Connection. This key would be saved in two pairing devices for future used.

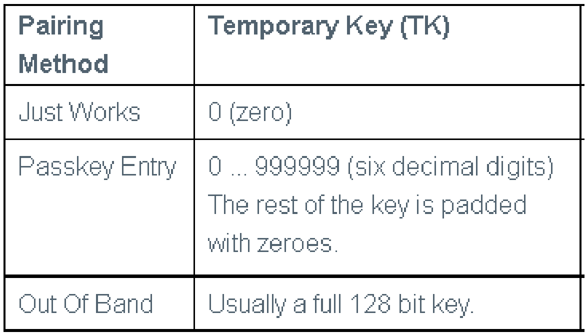
- Short-Term Key (STK) is used to encrypt a connection when 2 devices are pairing for the first time. LTK is transmitted through the encrypted connection.

- Temporary Key (TK) is used to calculate STK.

STK = AES128 (TK, Srand || Mrand), where Srand and Mrand are random numbers generated by two pairing devices (the Master and the Slave) for every connection

So we wanted to know how to calculate TK, from which we would be able to find STK and LTK respectively. From BTLE specifications, we found that the value of TK would be determined by the authentication type of the connection.

Back to our first approach, we discovered that the authentication type was decided based on I/O capabilities and therefore would be "Numeric Comparison" type. This lead to knowledge in the value of TK:



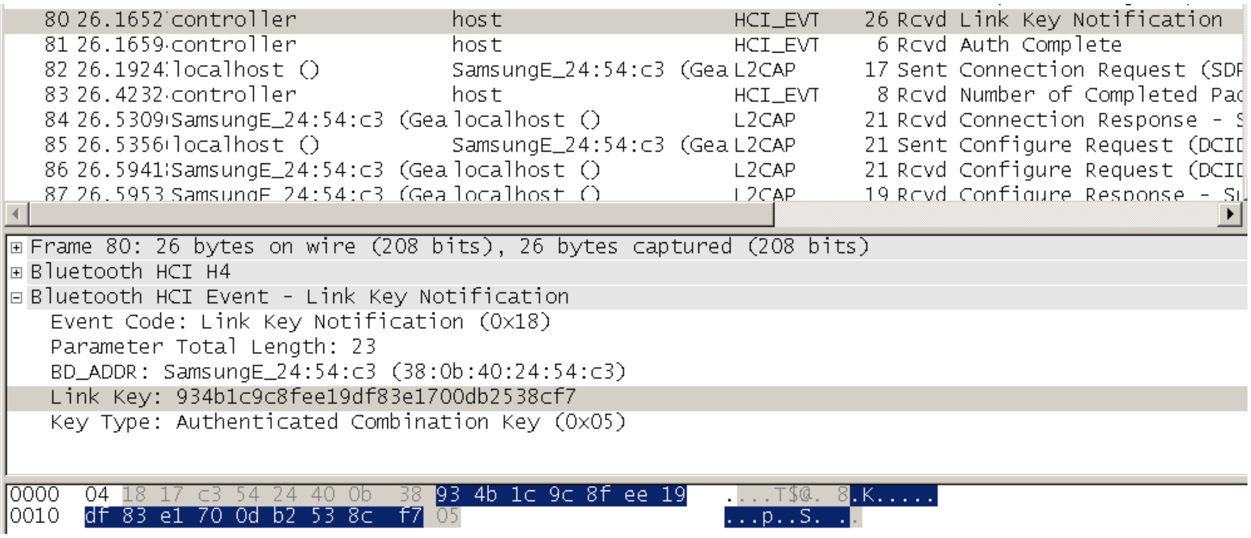
Pairing Method	Temporary Key (TK)
Just Works	0 (zero)
Passkey Entry	0 ... 999999 (six decimal digits) The rest of the key is padded with zeroes.
Out Of Band	Usually a full 128 bit key.

Relation between the TK and the pairing methods- Figure taken from [3]

Up to this point, we theoretically knew how the value of TK were obtained, and therefore the values of STK and LTK respectively. Our following steps were to study the risk of this key distribution and try to apply them. Let us show how far we progressed in the rest of this approach.

3.3. Potential Risks

- Risk we found in Bluetooth paper [6]: TK would be brute-forced using related values which transmitted in plaintext over the air. Then STK and LTK would be found respectively.
- Risk we found in our first approach (HCI logs approach): LTK could be found directly from the HCI logs.



```
80 26.1652 controller              host                     HCI_EVT    26 Rcvd Link Key Notification
81 26.1659 controller              host                     HCI_EVT     6 Rcvd Auth Complete
82 26.1924 localhost ()            SamsungE_24:54:c3 (Gea L2CAP      17 Sent Connection Request (SDF
83 26.4232 controller              host                     HCI_EVT     8 Rcvd Number of Completed Pac
84 26.5309 SamsungE_24:54:c3 (Gea localhost ()            L2CAP      21 Rcvd Connection Response - S
85 26.5356 localhost ()            SamsungE_24:54:c3 (Gea L2CAP      21 Sent Configure Request (DCID
86 26.5941 SamsungE_24:54:c3 (Gea localhost ()            L2CAP      21 Rcvd Configure Request (DCID
87 26.5953 SamsungE_24:54:c3 (Gea localhost ()            L2CAP      19 Rcvd Configure Response - Su

⊞ Frame 80: 26 bytes on wire (208 bits), 26 bytes captured (208 bits)
⊞ Bluetooth HCI H4
⊟ Bluetooth HCI Event - Link Key Notification
    Event Code: Link Key Notification (0x18)
    Parameter Total Length: 23
    BD_ADDR: SamsungE_24:54:c3 (38:0b:40:24:54:c3)
    Link Key: 934b1c9c8fee19df83e1700db2538cf7
    Key Type: Authenticated Combination Key (0x05)

0000  04 18 17 c3 54 24 40 0b  38 93 4b 1c 9c 8f ee 19   .....T$@. 8.K.....
0010  df 83 e1 70 0d b2 53 8c  f7 05                     ...p..S. ..
```

LTK could be obtained from Link_Key_Notification event in HCI logs

Both these risks would lead to an expose of Long-Term Key in the end.

3.4. OK, but why our attempts were not successful?

Our idea was to capture Long-term Key via HCI logs and other data packets via Ubertooth-One (channel 39). Then we would use the long-term key to decrypt these packets. These data packets, starting after CONNECT_REQ packet, were supposed to be encrypted using the Long-term Key obtained in HCI logs.

But when decrypting these packets, we got errors:

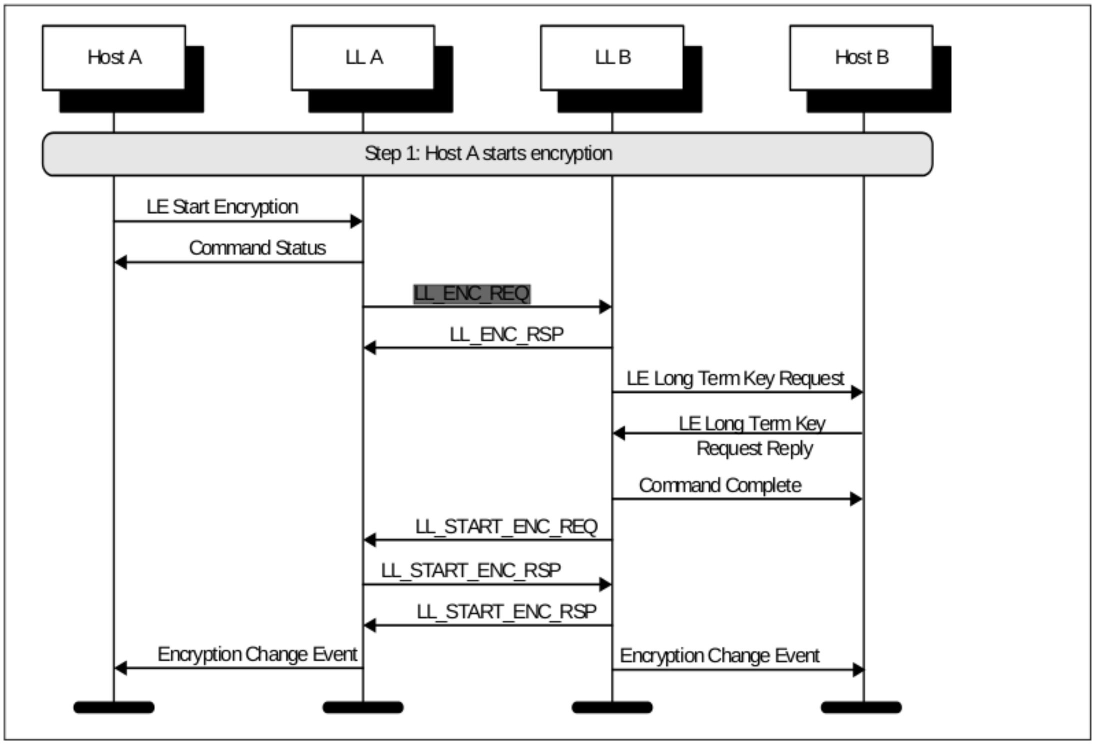
"No LL_ENC_REQ found

No LL_ENC_RSP found

Giving up due to 2 errors"

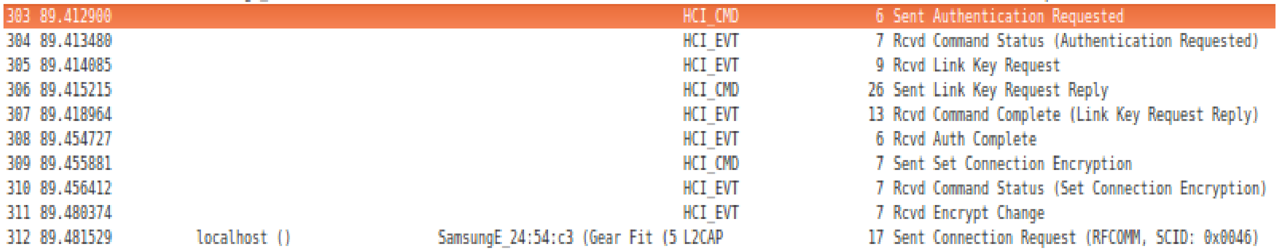
So we tried to understand why these packets were not captured by Ubertooth One and finally found a reason.

This is what expected for Bluetooth Low Energy:



Start Encryption in BTLE - Figure taken from [1],Vol 6, page 114 of 138

This is what actually happened and observed by HCI logs:

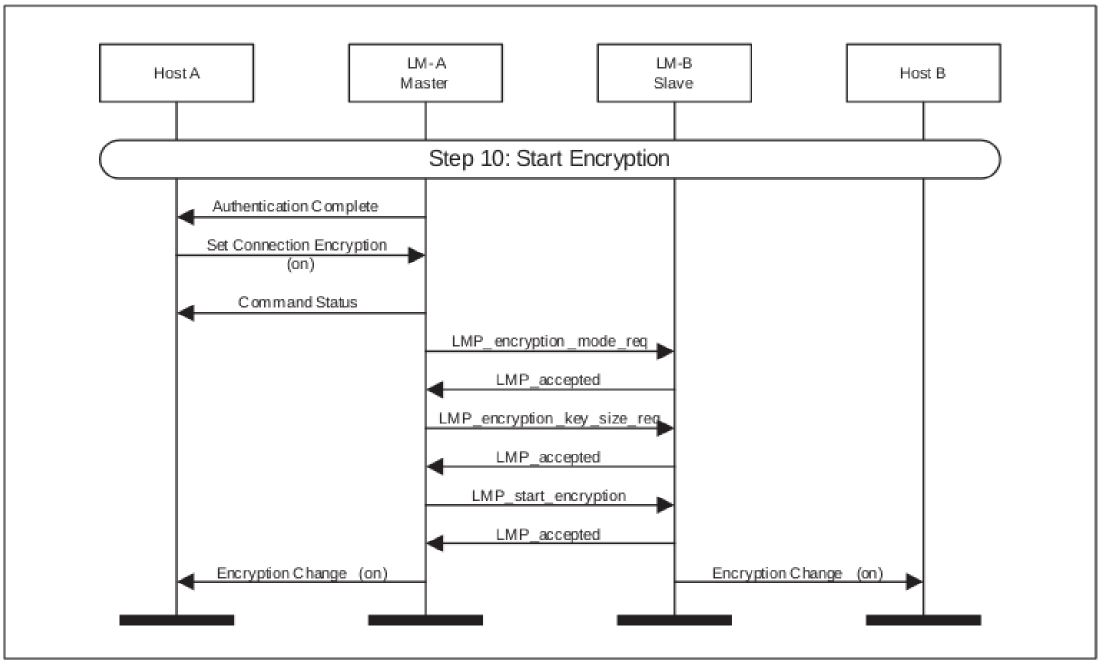


```
303 89.412900                                          HCI_CMD            6 Sent Authentication Requested
304 89.413480                                          HCI_EVT            7 Rcvd Command Status (Authentication Requested)
305 89.414085                                          HCI_EVT            9 Rcvd Link Key Request
306 89.415215                                          HCI_CMD           26 Sent Link Key Request Reply
307 89.418964                                          HCI_EVT           13 Rcvd Command Complete (Link Key Request Reply)
308 89.454727                                          HCI_EVT            6 Rcvd Auth Complete
309 89.455881                                          HCI_CMD            7 Sent Set Connection Encryption
310 89.456412                                          HCI_EVT            7 Rcvd Command Status (Set Connection Encryption)
311 89.480374                                          HCI_EVT            7 Rcvd Encrypt Change
312 89.481529       localhost ()       SamsungE_24:54:c3 (Gear Fit (5 L2CAP  17 Sent Connection Request (RFCOMM, SCID: 0x0046)
```

Set_Connection_Encryption command in HCI logs

“LL_ENC_REQ” and “LL_ENC_RSP” packets could only be sent by the Link Layer after HCI command “LE Start Encryption”. But here we did not find the dedicated command.

The observed log matched with the following schema which, surprisingly, was from BR/EDR (Bluetooth classic, not Bluetooth Low Energy):



Enable Encryptionin Bluetooth Classic - Figure taken from [1],Vol 2,page 906 of 1114

Things became more transparent here. The reason why, by using Ubertooth One, we did not obtain the two essential packets for encryption/decryption in Bluetooth Low Energy was that they had never been sent. The HCI command "Set Connection Encryption", which started the encryption process, was actually belong to BluetoothClassic (BR/EDR).

3.4.1. <u>Conclusions</u>

Consequently, we jumped into a conclusion that the Samsung smart watch, despite of being introduced as a Bluetooth Low Energy device, actually employed Bluetooth Classic's Link Manager Protocol as its method to establish an encryption when communicating with Samsung specific application Gear Fit Manager. This made potential attacksinto communication of the watch, which targeted the air interface and potential risks of Bluetooth Low Energy, become less feasible than expected.

About Bluetooth Classic security risks, please find more information in next section "Additional findings". This could be a potential topic for future research.

3.5. Additional findings

In this section, we introduced some related facts we found about Bluetooth Classic and the communication of the Samsung smart watch. They might not help us to reach our goals, but still be interesting or useful for future reference.

3.5.1. Ubertooth One Hardware supports Bluetooth Classic sniffing?

According to Ubertooth One Project [4], the hardware contains a Texas Instrument chip CC2400[11], which can be used as 2.4 GHz radio frequency transceiver for wireless applications.

Taking into account that the Bluetooth Classic frequency also occupies a section of the 2.4 GHz band, we concluded Ubertooth One Hardware would support Bluetooth Classic sniffing.

3.5.2. Existing Bluetooth Classic sniffing projects?

There are projects of other people who are also working on the sniffing of Bluetooth Classic. However, as far as we have found, there is no project which succeeds in completely eavesdroppingentire data from Bluetooth Classic connection.

Their most difficult problem has been following the hopping frequency of Bluetooth Classic. Unlike Bluetooth Low Energy, Bluetooth Classic has a complex structure of hopping frequency. Therefore, projects which use either Ubertooth One hardware [7] or using Universal Software Radio Peripheral (USRP)[8] are lack of frequency hopping supports.

Although there have been many obstacles, these projects have succeeded in sniffingsome necessary parameters for following Bluetooth Classic frequency hopping pattern. Consequently, potential Bluetooth Classing sniffing devices are likely to be created in future.

3.5.3. Bluetooth Classic and Bluetooth LE interoperability?

According to Bluetooth standards [12], Bluetooth Low Energy would contain two modes:

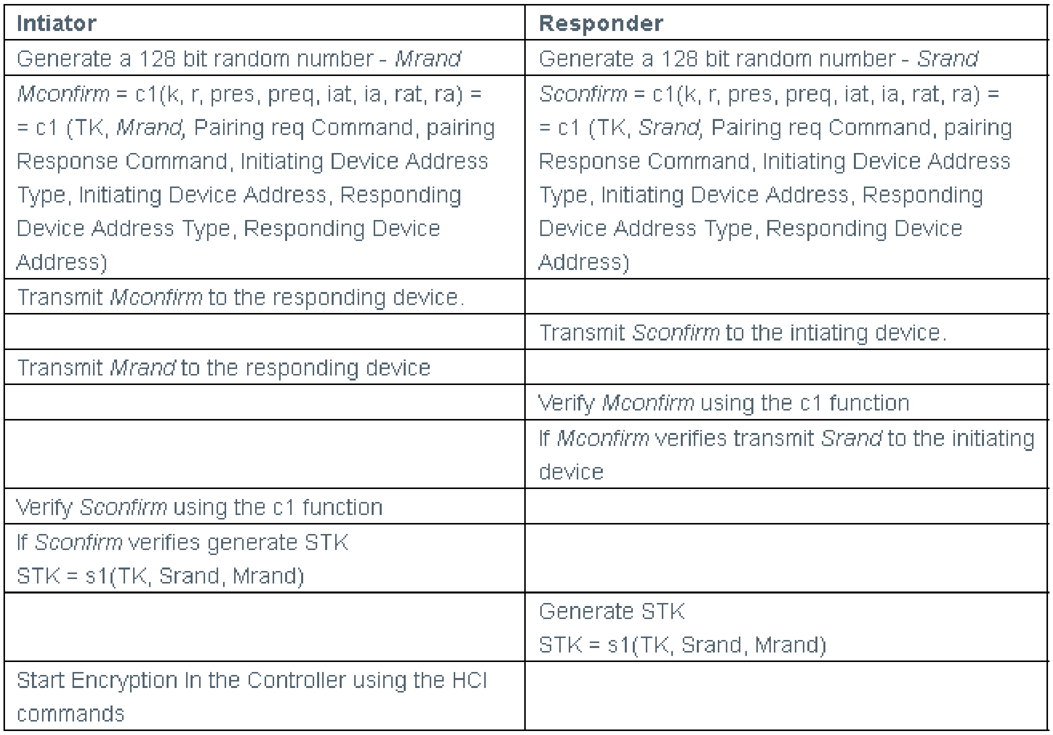
singlemode and dual mode. In single mode, only Bluetooth Low Energy is used while dual modecan operate on both Bluetooth Classic and Bluetooth Low Energy.

Below is a table we made in order to illustrate which devices/applications have used which types of Bluetooth.

Types	Requirements of Devices/Applications	Devices/Applications
Only Bluetooth Classic	Devices/Applications which require a stable data rate or quality transmission (like audio transmissions)	Wireless Headphones, Speakers, Audio headsets…
Single Mode Only Bluetooth Low Energy	Devices/Applications which require only transmitting smaller amounts of data and longer usage of battery cells	Heart-rate or blood pressure monitors, text transmitters of watches, tablets…
Dual Mode	Devices/Applications which require communications with other Devices/Applications using Bluetooth classic	Bluetooth Mouse, Keyboards…

A Survey of Bluetooth Classic and Bluetooth LE interoperability

3.5.4. Short-Term Key Calculation and Distribution



Intiator	Responder
Generate a 128 bit random number - *Mrand*	Generate a 128 bit random number - *Srand*
Mconfirm = c1(k, r, pres, preq, iat, ia, rat, ra) = = c1 (TK, *Mrand*, Pairing req Command, pairing Response Command, Initiating Device Address Type, Initiating Device Address, Responding Device Address Type, Responding Device Address)	*Sconfirm* = c1(k, r, pres, preq, iat, ia, rat, ra) = = c1 (TK, *Srand*, Pairing req Command, pairing Response Command, Initiating Device Address Type, Initiating Device Address, Responding Device Address Type, Responding Device Address)
Transmit *Mconfirm* to the responding device.	
	Transmit *Sconfirm* to the intiating device.
Transmit *Mrand* to the responding device	
	Verify *Mconfirm* using the c1 function
	If *Mconfirm* verifies transmit *Srand* to the initiating device
Verify *Sconfirm* using the c1 function	
If *Sconfirm* verifies generate STK STK = s1(TK, Srand, Mrand)	
	Generate STK STK = s1(TK, Srand, Mrand)
Start Encryption In the Controller using the HCI commands	

How to calculate STK when knowing TK - Figure taken from [3]

PART B: RISK IN FIRMWARE AND GEARFIT'S APPLICATION (Contributor – Kim Hung)

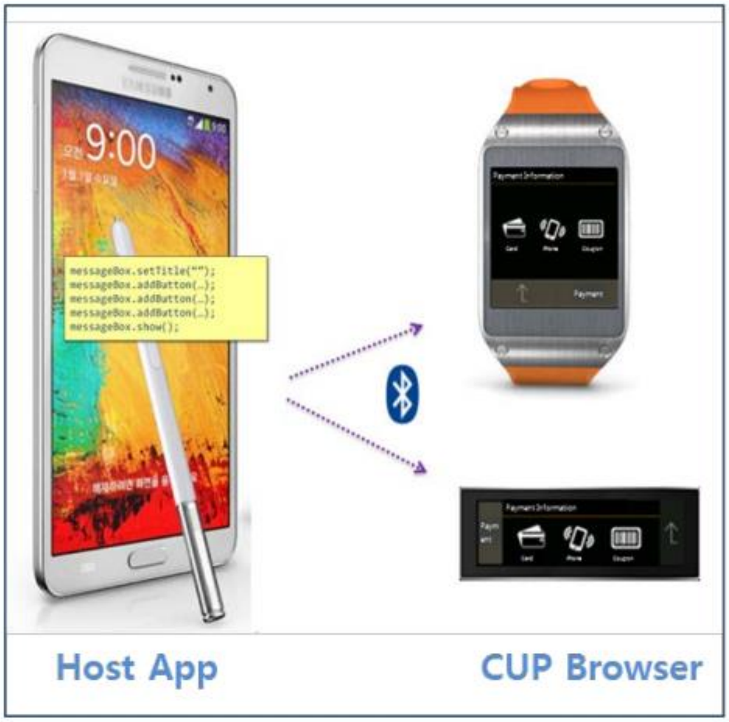
I. Android:

1. Samsung Smartwatch App using CUP:

1.1. Overview:

Companion UI Platform (CUP) is a open library which allows you to display information from a host device, for example a smartphone, on companion devices and devices that use CUP. CUP contains many control widgets named winsets, which is useful when you want to create layouts to display information.

The host device control the companion device's display by sending request to companion devices using Bluetooth. CUP Browser is facilitate to display the CUP winset on wearable devices in many types, and receive interaction events back from the wearable devices.



```
messageBox.setTitle("");
messageBox.addButton(…);
messageBox.addButton(…);
messageBox.addButton(…);
messageBox.show();
```

Figure 1: CUP overview [16]

1.2 Architecture:

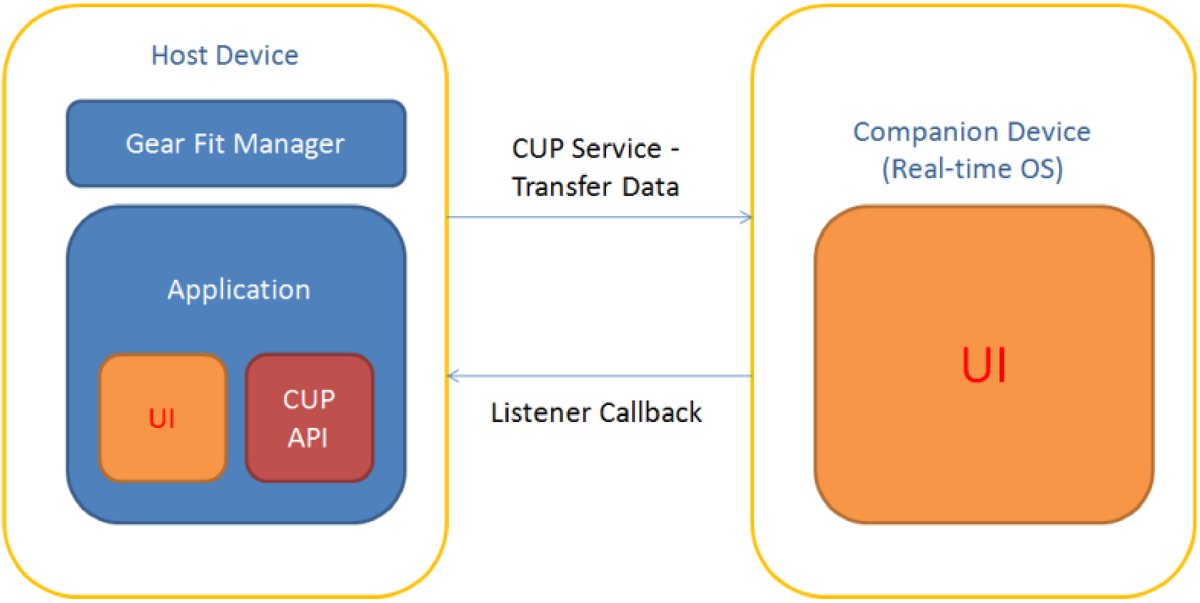


Figure 2: CUP Architecture [16]

The CUP architecture is simply contained:

- Applications: The applications are built on CUP as main platform.
- CUP API: Components for creating and showing various layouts on the companion device, including the callback interface.
- CUP Service: Service used to connect between the host and companion device.

2. CUP library:

2.1. Cup Technology:

The purpose of CUP is to provide interaction between a hosting Android device and its wearable CUP Browsers.

- CUP Host consists of the host application built from CUP SDK, and the classes that show more than 12 wingets on a CUP Browser.
- CUP Browser has responsible to receive and perform host request. The CUP winset displays the commands on the CUP Browser and sends back interactive events to the CUP host.

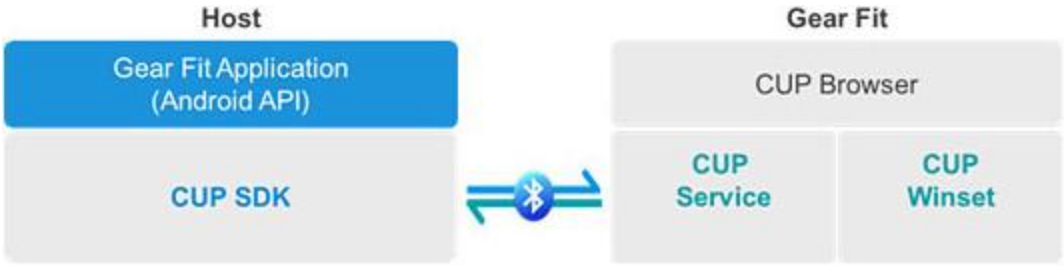


Figure 3: CUP Architecture [16]

The CUP process functions as follows:

- The host application sends a command to the CUP browser requiring the Browser to display a certain winset.
- The CUP Browser displays the winset.
- After the user interacts with the winset, the Browser sends the user event back to the host application, which can proceed to the next step.

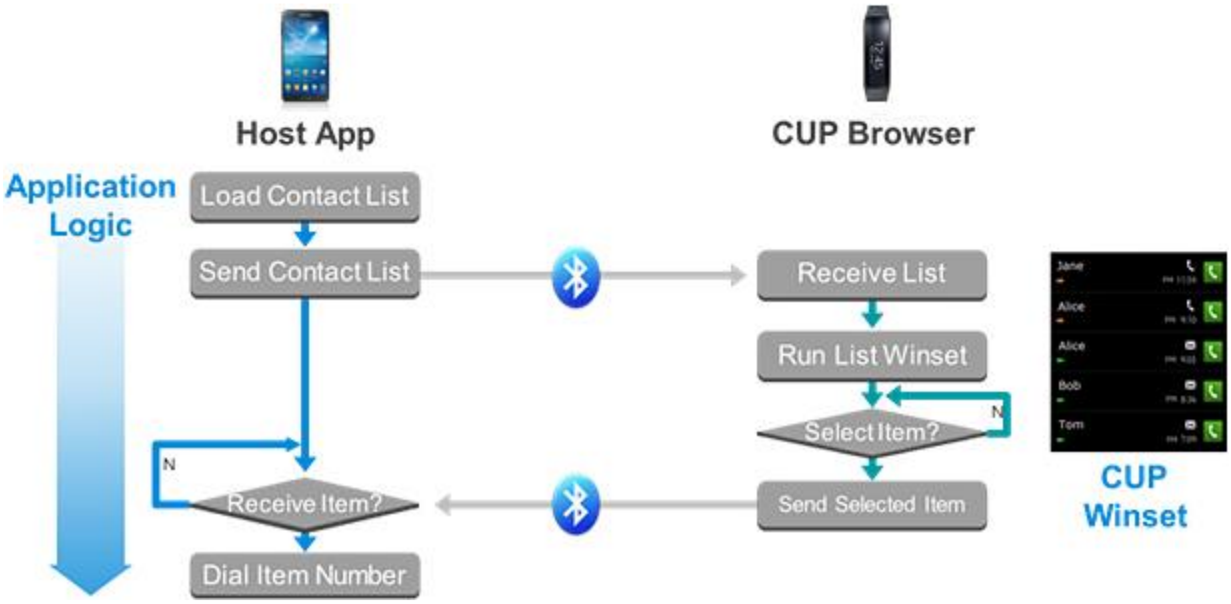


Figure 4: CUP Interaction [16]

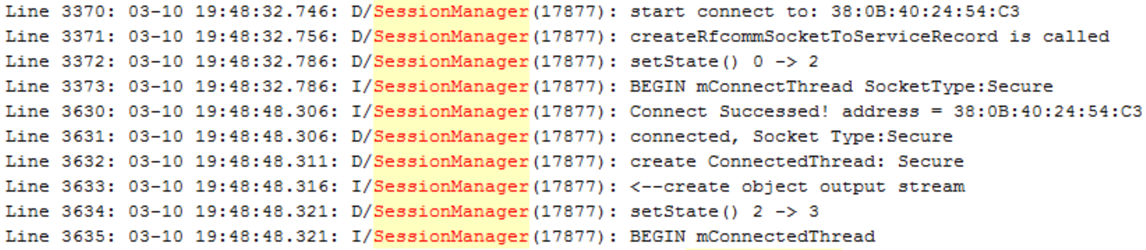
2.2. Scup Library Components:

The CUP classes include some important elements:

- Scup: Initializes CUP.
- ScupWidgetBase: Provides basic widget functions in companion devices.
- ScupButton: Provides clickable buttons for the companion UI, with a callback listener for the button clicks.
- ScupDialog: Supports displaying a screen on the companion device. Each dialog can contain multiple widgets.
- ScupLabel: Shows information and images.
- ScupListBox: Shows a list of information.
- ScupSpinner: Shows the same spinner as in Android.

3. Bluetooth Connecting:

To identify Bluetooth connection process between android smartphone and gearfit. We use android system log which is collected while connecting.

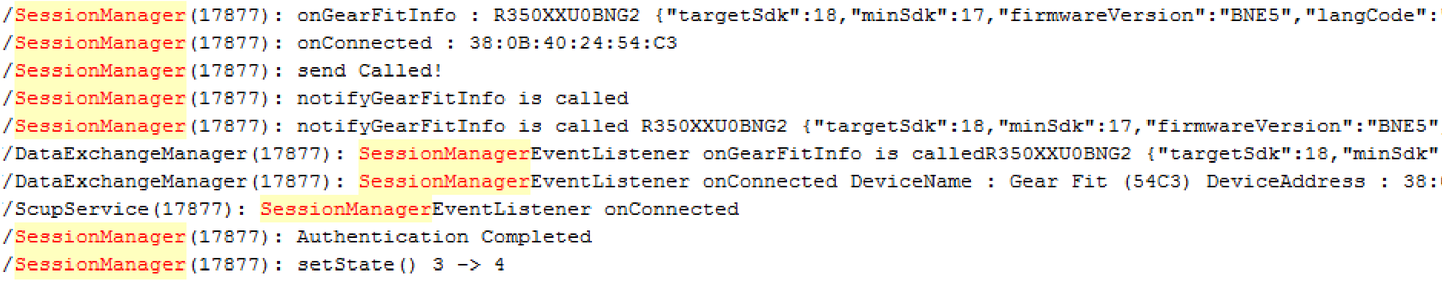


```
Line 3370: 03-10 19:48:32.746: D/SessionManager(17877): start connect to: 38:0B:40:24:54:C3
Line 3371: 03-10 19:48:32.756: D/SessionManager(17877): createRfcommSocketToServiceRecord is called
Line 3372: 03-10 19:48:32.786: D/SessionManager(17877): setState() 0 -> 2
Line 3373: 03-10 19:48:32.786: I/SessionManager(17877): BEGIN mConnectThread SocketType:Secure
Line 3630: 03-10 19:48:48.306: I/SessionManager(17877): Connect Successed! address = 38:0B:40:24:54:C3
Line 3631: 03-10 19:48:48.306: D/SessionManager(17877): connected, Socket Type:Secure
Line 3632: 03-10 19:48:48.311: D/SessionManager(17877): create ConnectedThread: Secure
Line 3633: 03-10 19:48:48.316: I/SessionManager(17877): <--create object output stream
Line 3634: 03-10 19:48:48.321: D/SessionManager(17877): setState() 2 -> 3
Line 3635: 03-10 19:48:48.321: I/SessionManager(17877): BEGIN mConnectedThread
```

Figure 5: Bluetooth Log State 1-2

At the beginning, Bluetooth set up a normal Bluetooth connection between 2 devices via RFCOMM channels using UUID equal "9c86c750-870d-11e3-baa7-0800200c9a66". After success creating channel, 2 devices perform authentication state:

- Gearfit send message contain its information: R350XXU0BNG2{"targetSdk":18,"minSdk":17,"firmwareVersion":"BNE5","langCode":"vi","localeCode":"vi-VN","modelNumber":"SMR350","firmwareFilename":"UPDATEMODE_R350OXA0BNE5_R350XXU0BNE5.tar"} XEF RFAF40215TF
- Phone received this information and check geafit firmware need update or not. It will send query result to gearfit to finish authentication state.



```
/SessionManager(17877): onGearFitInfo : R350XXU0BNG2 {"targetSdk":18,"minSdk":17,"firmwareVersion":"BNE5","langCode":
/SessionManager(17877): onConnected : 38:0B:40:24:54:C3
/SessionManager(17877): send Called!
/SessionManager(17877): notifyGearFitInfo is called
/SessionManager(17877): notifyGearFitInfo is called R350XXU0BNG2 {"targetSdk":18,"minSdk":17,"firmwareVersion":"BNE5"
/DataExchangeManager(17877): SessionManagerEventListener onGearFitInfo is calledR350XXU0BNG2 {"targetSdk":18,"minSdk"
/DataExchangeManager(17877): SessionManagerEventListener onConnected DeviceName : Gear Fit (54C3) DeviceAddress : 38:
/ScupService(17877): SessionManagerEventListener onConnected
/SessionManager(17877): Authentication Completed
/SessionManager(17877): setState() 3 -> 4
```

Figure 6: Bluetooth Log State 3-4

4. Gear fit installing process:

Base on the log file dump from android phone via adb process and reverse APK file. Filter some critical information about the Scup service which is the main service using to control the installing application process. We built the process:

INSTALL APPLICATION PROCCESS

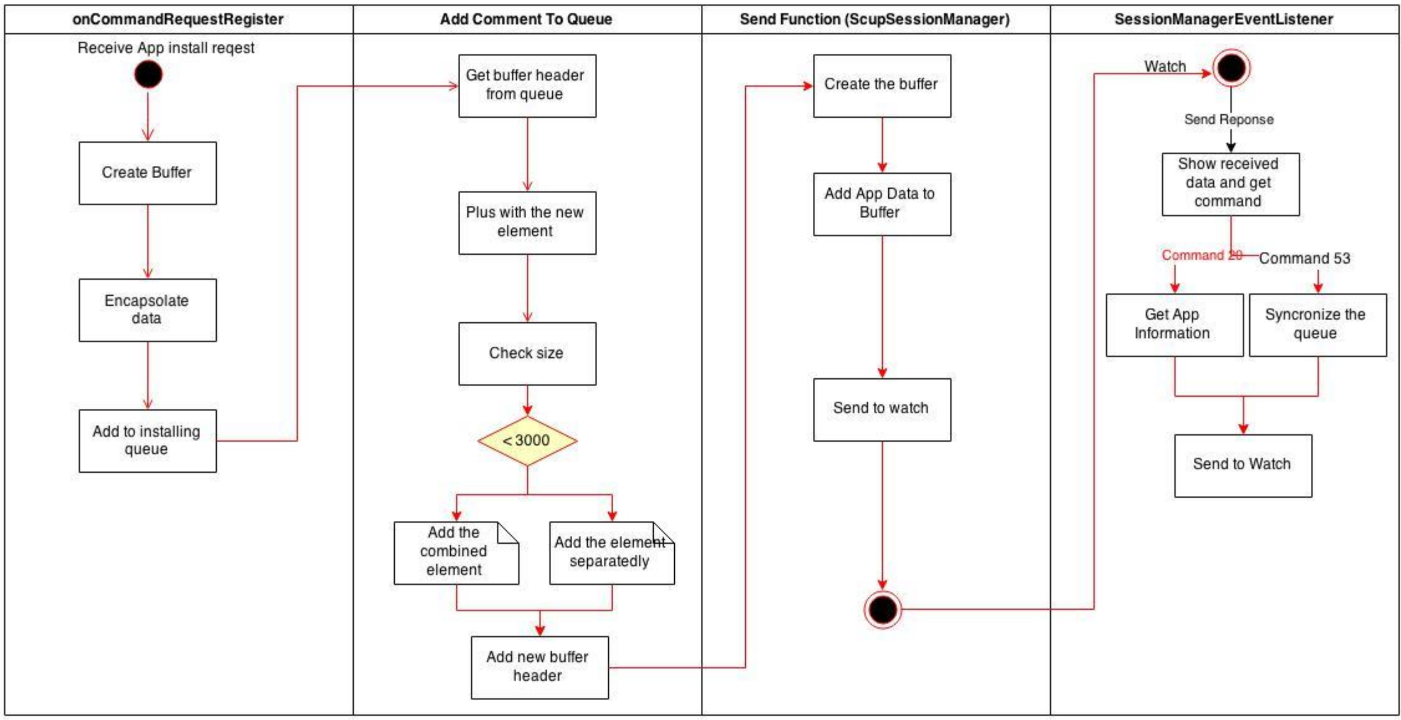


Figure 7: Install Application Process

II. Reverse APK file:

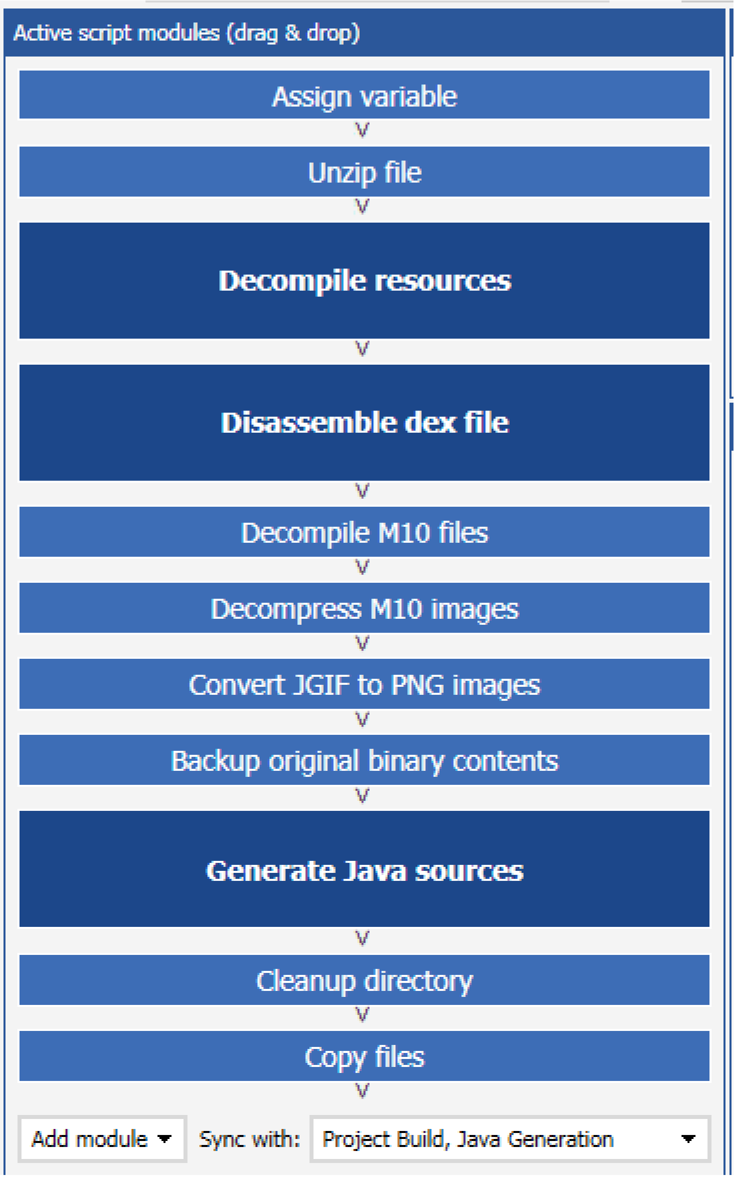
1. APK reverse process:

1.1. Basic approach:

We use dex2jar tool to generate Java Code from unknown APK file. Once JAR is generate, we use JD-GUI which is a standalone graphical utility that displays Java source codes of ".class" files. You can browse the reconstructed source code with the JD-GUI for instant access to methods and fields.

1.2.Virtuous Ten Studio:

Virtuous Ten Studio (*VTS*) is a free application to reverse engineering android applications. This program allows you to manage entire Android projects within an easy to use and familiar environment. With this application we can easily decompile, edit and recompile any apk or jar file.



Active script modules (drag & drop)

Assign variable
V
Unzip file
V
Decompile resources
V
Disassemble dex file
V
Decompile M10 files
V
Decompress M10 images
V
Convert JGIF to PNG images
V
Backup original binary contents
V
Generate Java sources
V
Cleanup directory
V
Copy files
V

Sync with: Project Build, Java Generation

Figure 8: APK reverse process

2. <u>Android application Log:</u>

To connect to android device, we use Android Debug Bridge (adb) is a versatile command line tool that is a client-server program that includes three components:

- A client, which runs on your development machine. You can invoke a client from a shell by using an adb command. Other Android tools such as the ADT plugin and DDMS also create adb clients.
- A server, which runs as a background process on your development machine. The server manages communication between the client and the adb daemon running on an emulator or device.
- A daemon, which runs as a background process on device instance.

After connect successfully, we use Android logging system which provides a mechanism for collecting and viewing system debug output. Logs from various applications and portions of the system are collected in a series of circular buffers, which then can be viewed and filtered by the logcat command. You can use logcat from an ADB shell to view the log messages.

III. Reverse Firmware:

1. Hardware overview:

Here are overview about external GearFit component:

- 1.84" Curved Super AMOLED touchscreen display (432 x 128 pixels)
- 180 MHz ARM Cortex-M4 CPU
- Accelerometer, gyroscope, and heart rate sensor
- Battery good for 3-4 days of normal use
- Bluetooth 4.0 LE

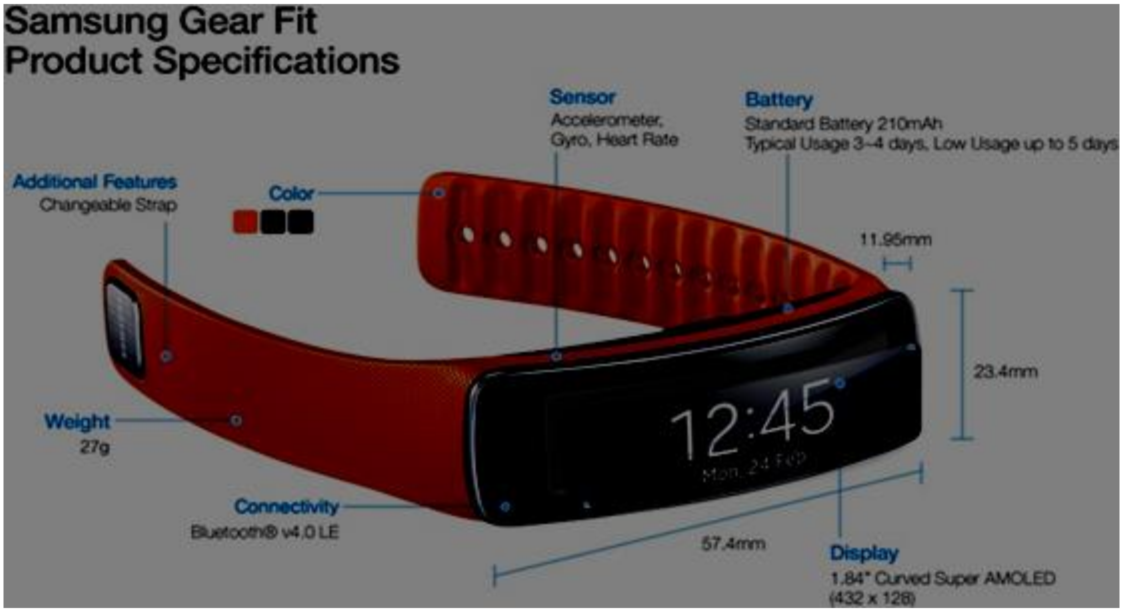


Figure 9 : Gear Fit Product Specification [20]

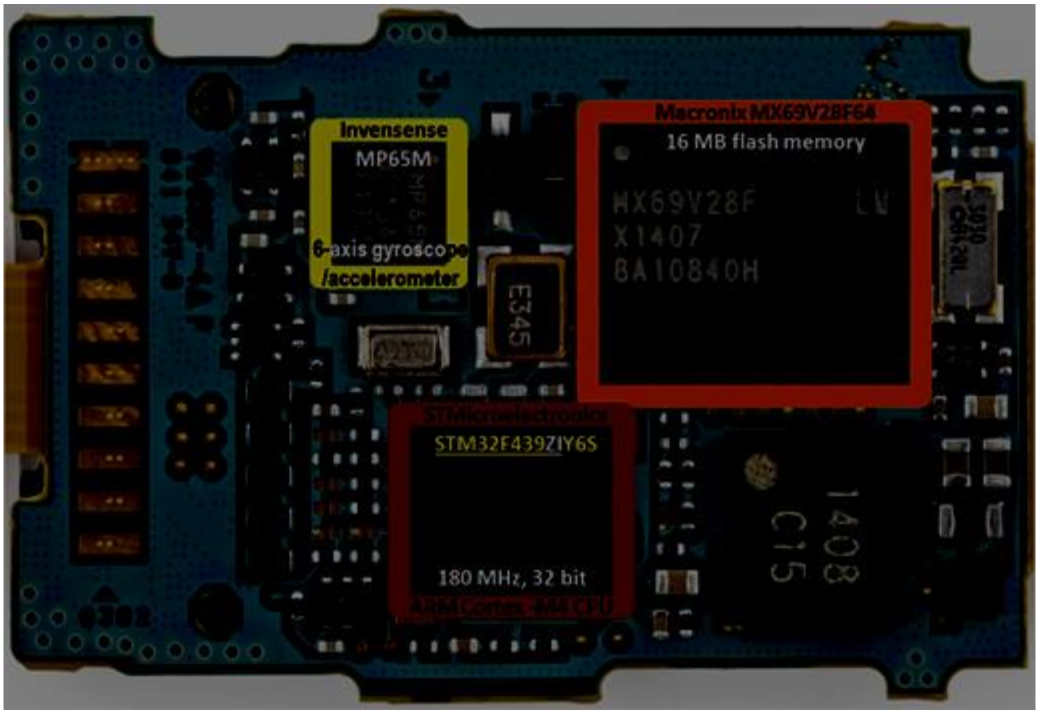


Figure 10: Gear Fit main board [20]

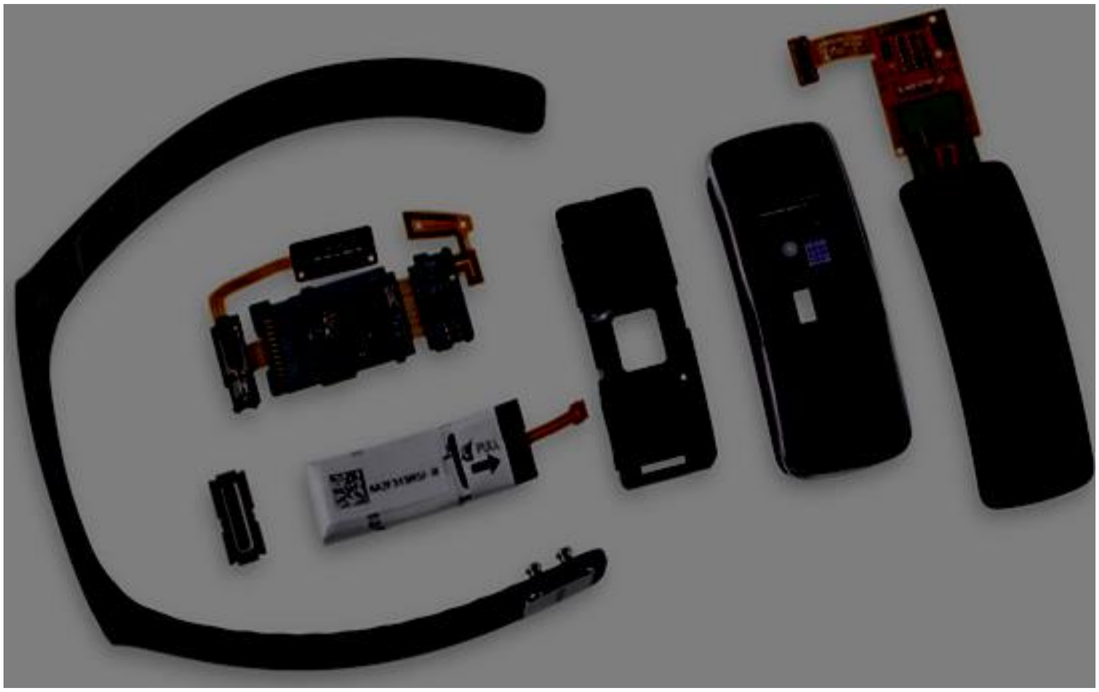
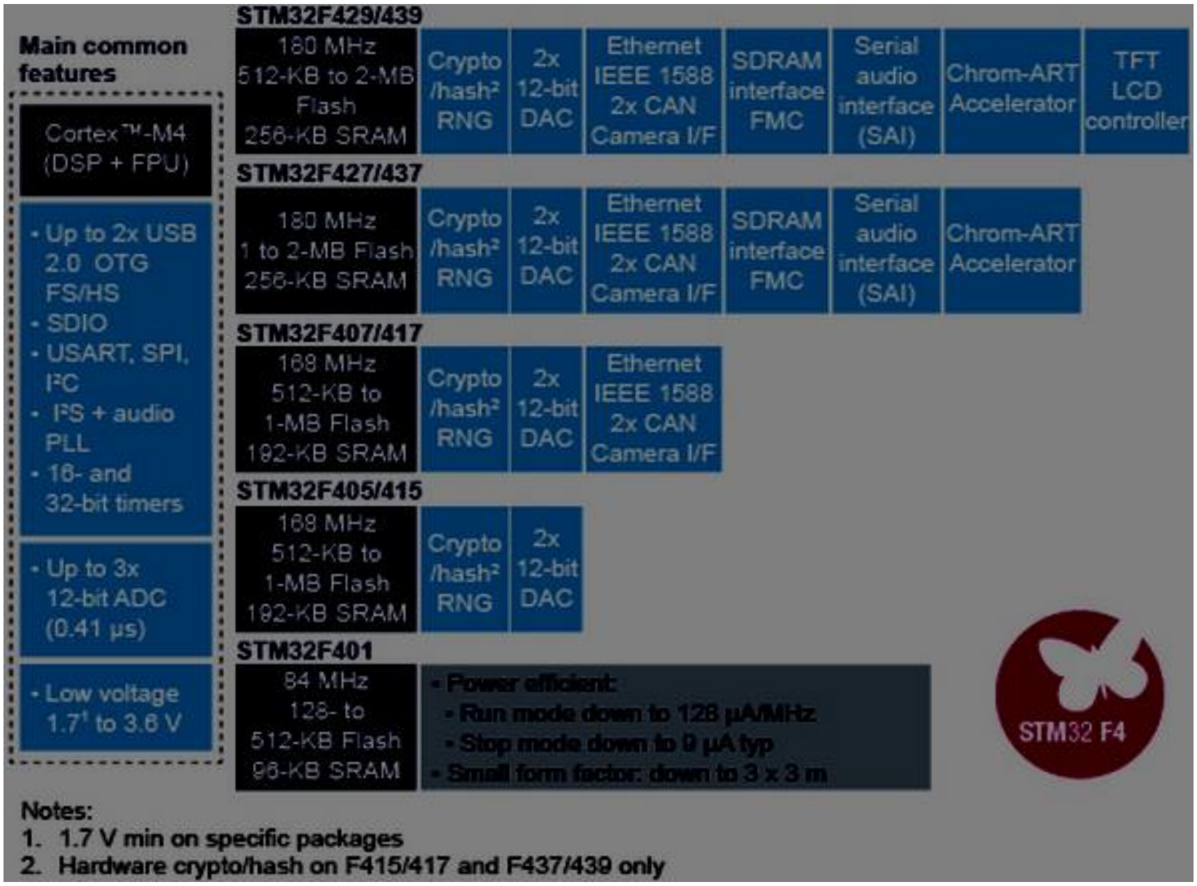


Figure 11: Hardware main component [20]

Only the following key components are considered here:

- The STM32F439ZIY6S High-performance Microcontroller of the STM32 F4 Series MCUs from STMicroelectronics

- The MPU-6500 Six-Axis (Gyro + Accelerometer) MEMS Motion Tracking™ Devices from InvenSense.
- BCM4334WKUBG Single-Chip Dual-Band Combo Device Supporting 802.11n, Bluetooth 4.0+HS & FM Receiver from Broadcom



Main common features

- Cortex™-M4 (DSP + FPU)
- Up to 2x USB 2.0 OTG FS/HS
- SDIO
- USART, SPI, I²C
- I²S + audio PLL
- 16- and 32-bit timers
- Up to 3x 12-bit ADC (0.41 µs)
- Low voltage 1.7¹ to 3.6 V

STM32F429/439	180 MHz 512-KB to 2-MB Flash 256-KB SRAM	Crypto /hash² RNG	2x 12-bit DAC	Ethernet IEEE 1588 2x CAN Camera I/F	SDRAM interface FMC	Serial audio interface (SAI)	Chrom-ART Accelerator	TFT LCD controller
STM32F427/437	180 MHz 1 to 2-MB Flash 256-KB SRAM	Crypto /hash² RNG	2x 12-bit DAC	Ethernet IEEE 1588 2x CAN Camera I/F	SDRAM interface FMC	Serial audio interface (SAI)	Chrom-ART Accelerator	
STM32F407/417	168 MHz 512-KB to 1-MB Flash 192-KB SRAM	Crypto /hash² RNG	2x 12-bit DAC	Ethernet IEEE 1588 2x CAN Camera I/F				
STM32F405/415	168 MHz 512-KB to 1-MB Flash 192-KB SRAM	Crypto /hash² RNG	2x 12-bit DAC					
STM32F401	84 MHz 128- to 512-KB Flash 96-KB SRAM	- Power efficient: - Run mode down to 128 µA/MHz - Stop mode down to 9 µA typ - Small form factor: down to 3 x 3 m						

STM32 F4

Notes:
1. 1.7 V min on specific packages
2. Hardware crypto/hash on F415/417 and F437/439 only

Figure 12: Chip compare version [20]

2. IDA plugin:

Although our efforts have improved IDA's initial analysis, there is still a good deal of code that has been missed. We have to write some simple IDA scripts which can be used to get more out of the disassembly. First, we want to locate unidentified functions by iterating through the code looking for common function prologues. If one is found, we'll tell IDA to create a function there.

IDAPython is an IDA Pro plugin that integrates the Python programming language, allowing scripts to run in IDA Pro. These programs have access to IDA Plugin API, IDC and all modules available for Python. The power of IDA Pro and Python provides a platform for easy prototyping of reverse engineering and other research tools. Our IDAPython script will search the code (starting at the cursor position) for byte sequences that correspond to these instructions, and instruct IDA to convert them to functions.

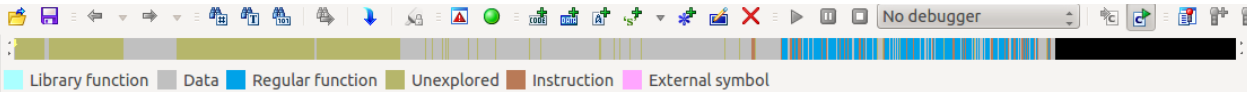
2.1.Find function:

This Script use to find functions which is not recognized by IDA. We use “BADADD” to definite about address that is valid or not. The script simply browse to all address and make the function if this addess is a “BADADD” and do not have function name.

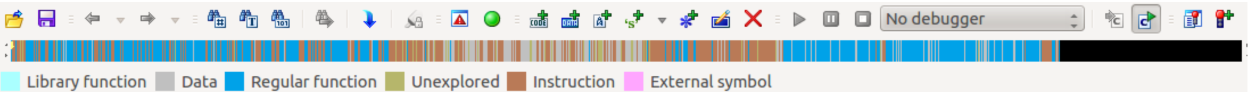
Main function is called from IDC library:

– idc.GetSegmentAttr(address, attr): Use to get the attribute of segment code at specific address.
– idc.GetFunctionName(address): Use to get function name of function which contains specific addess.
– idc.MakeFunction(address): Create function at specific address.
– idc.NextAddr(address): Move to next address.
Running on Wingtip firmware archive a good analyzing:

Before running script:



After running script:



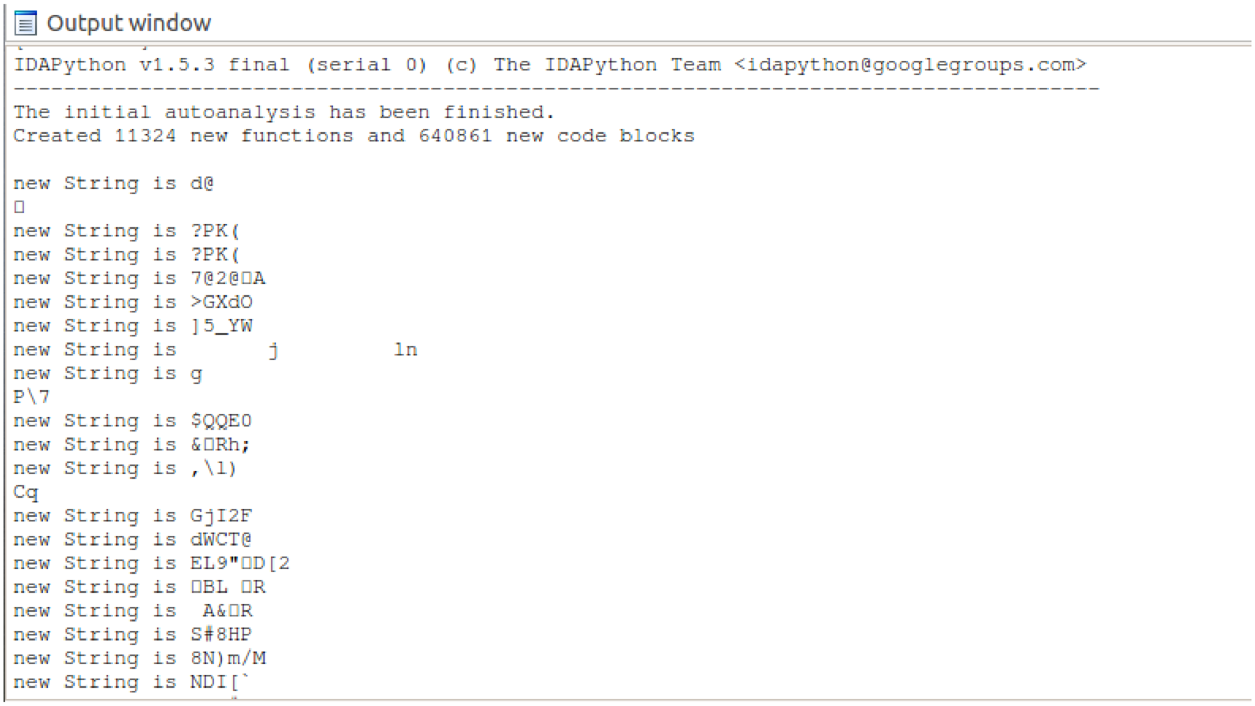
2.2.Find String:

This Script use resemble method which use “BADADDR” to definite about address that is valid or not. The script simply browse to all String in the list and create new string if it satisfy 3 conditions:

– Address is higher than first address.
– It is not ASCII.
– We can create new string fromcurrent address to the end String (calculated by OS).
It is also cover remaining data into DWORD by using idc.OpOff and idc.MakeDword functions.

Main function is called from IDC library:

– idc.MakeStr(address, end_address): Use to create string from address to the end address. If end_address specify to be BADADDR, end_address is calculated by OS.
– idc.isASCII(address): Check String at address is ASCII or not.
– idautils.Strings(): Return the list of valible String.
– idc.GetFlags(address): Get 32-bit value of internal flags.
– idc.MakeDword(address): Cover current item to Dword.
– idc.OpOff(ea, n, base): Convert operand to an offset
Our script can found more String but it do not bring to much valuable:



```
IDAPython v1.5.3 final (serial 0) (c) The IDAPython Team <idapython@googlegroups.com>
--------------------------------------------------------------------------------------
The initial autoanalysis has been finished.
Created 11324 new functions and 640861 new code blocks

new String is d@
□
new String is ?PK(
new String is ?PK(
new String is 7@2@□A
new String is >GXdO
new String is ]5_YW
new String is       j         ln
new String is g
P\7
new String is $QQE0
new String is &□Rh;
new String is ,\l)
Cq
new String is GjI2F
new String is dWCT@
new String is EL9"□D[2
new String is □BL □R
new String is  A&□R
new String is S#8HP
new String is 8N)m/M
new String is NDI[`
```

Figure 13: Script result

3. Firmware Analize:

Identify the loading address for the executable code: Loading the boot loader at address 0, Architecture ARM, the first few lines of firmware:

```
ROM:00000000  dword_0     DCD  0x20002DE0
ROM:00000004              DCD  0x20016AAD
ROM:00000008              DCD  0x2000E241
ROM:0000000C              DCD  0x2000E245
ROM:00000010              DCD  0x2000E2B5
ROM:00000014              DCD  0x2000E32D
ROM:00000018              DCD  0x2000E3A5
ROM:0000001C              DCD  0
ROM:00000020              DCD  0
```

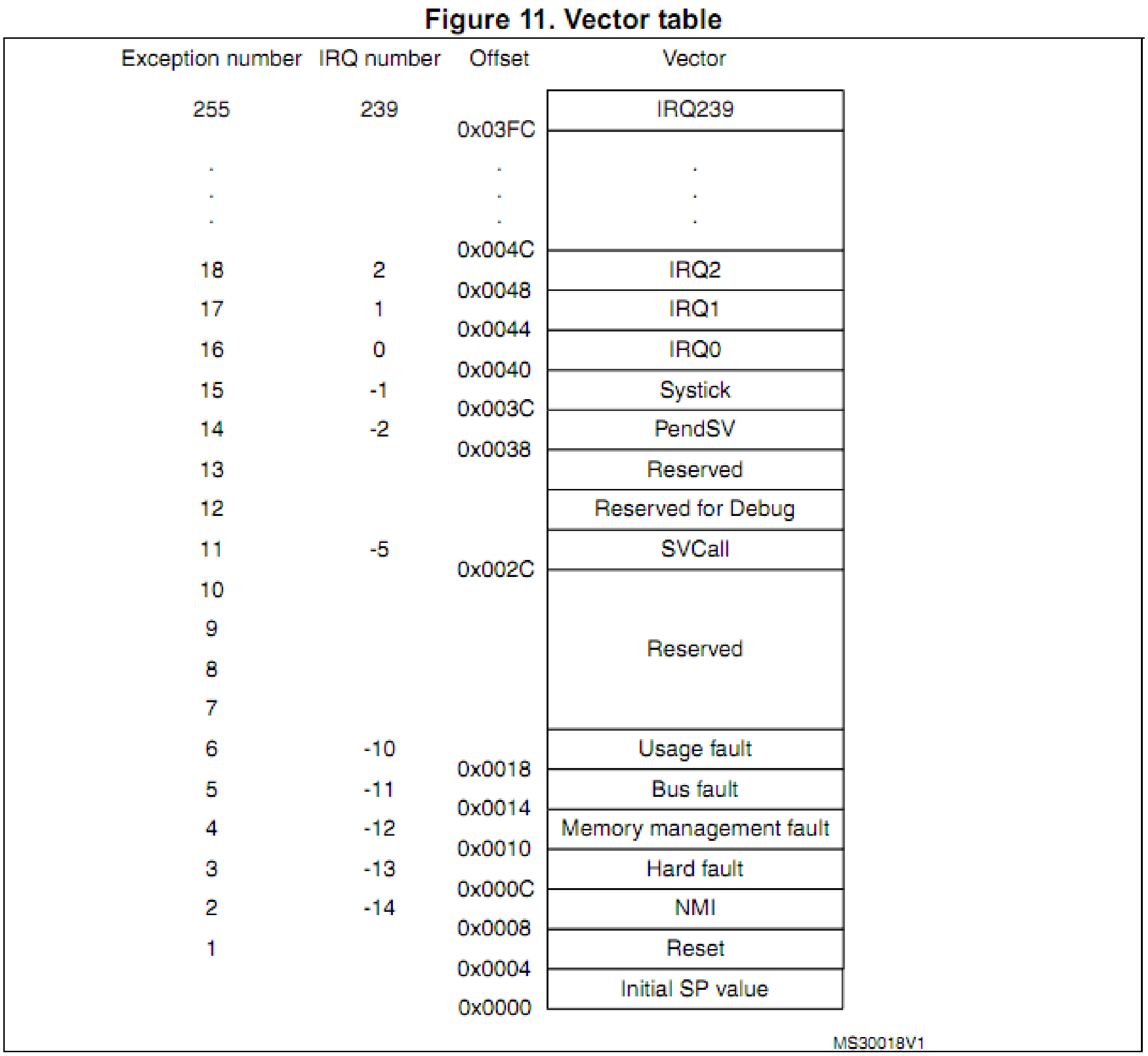


Figure 11. Vector table

Exception number	IRQ number	Offset	Vector
255	239	0x03FC	IRQ239
. . .		. . . 0x004C	. . .
18	2	0x0048	IRQ2
17	1	0x0044	IRQ1
16	0	0x0040	IRQ0
15	-1	0x003C	Systick
14	-2	0x0038	PendSV
13			Reserved
12			Reserved for Debug
11	-5	0x002C	SVCall
10 9 8 7			Reserved
6	-10	0x0018	Usage fault
5	-11	0x0014	Bus fault
4	-12	0x0010	Memory management fault
3	-13	0x000C	Hard fault
2	-14	0x0008	NMI
1		0x0004	Reset
		0x0000	Initial SP value

Figure 14: Vector table [14]

Compare with table vector of Smartwatch chip, we identify that: 0x20002DE0 is the initial stack pointer, and 0x20016AAD is the entry point.

That entry point does not seem to be contained in the boot loader itself (which has 64 K + 256 bytes), but in the on-Chip ROM.

In the modern complex chips such as STM32 F4 Series the first piece of code which is executed is usually not user code but the on-chip Boot ROM. It might check various conditions to determine where to load the user code from and whether to load it at all. In such cases, the user code might have to conform to different start-up conventions.

Unless you can somehow dump the memory at 0x20016AAD, you won't be able to find out what the ROM does, and where within the ROM it jumps. As this explanation, loading bootloader at 0x0 address is the best way in this situation.

4. Boot Sequence:

Step 1: The reset

On startup, the processor will jump to fixed location at address 0x0. This address should contain the reset vector and the default vector table. Reset vector is always the first instruction to be executed. The reset vector in this table will contain a branch to an address which will contain the reset code. Normally, at this stage, the rest of the vector table contains just the dummy handler a branch instruction that causes an infinite loop

Step 2: The reset code

This reset code to which the jump has been executed from the reset vector will do the following tasks:

- Set up system registers and memory environment
- Set up MMU
- Setup stack pointers : initialize stack pointers for all ARM modes
- Set up BSS section : zeroing the ZI data region, copying initialization values for initialized variables from ROM to RAM
- Set up hardware configuration: CPU clock initialization, external bus interface configuration, low level peripheral initialization.

Step 3: Remap Memory

One of the job of the initial reset code will be memory remapping. At the time of power up, the processor jumps at fixed location 0x0.So, this is important to ensure there is some executable code present at this location at the time of power up. And to ensure this, some non-volatile memory should be mapped to this address.

Vector table is located in RAM .However, ROM is located at 0x0 address and then during normal execution RAM is re-mapped to this location. Memory remapping can be achieved through hardware remapping, that is changing the address map of the bus. This can also be done through MMU.

Step 4: Setting up the external memory, loading and executing the OS image

External memory should be setup before loading an image to it (Refresh rate, clock…), OS image can then be loaded from flash to RAM. The OS image may be compressed in which case it needs to be decompressed before PC can be modified to point to the operating system entry point address.

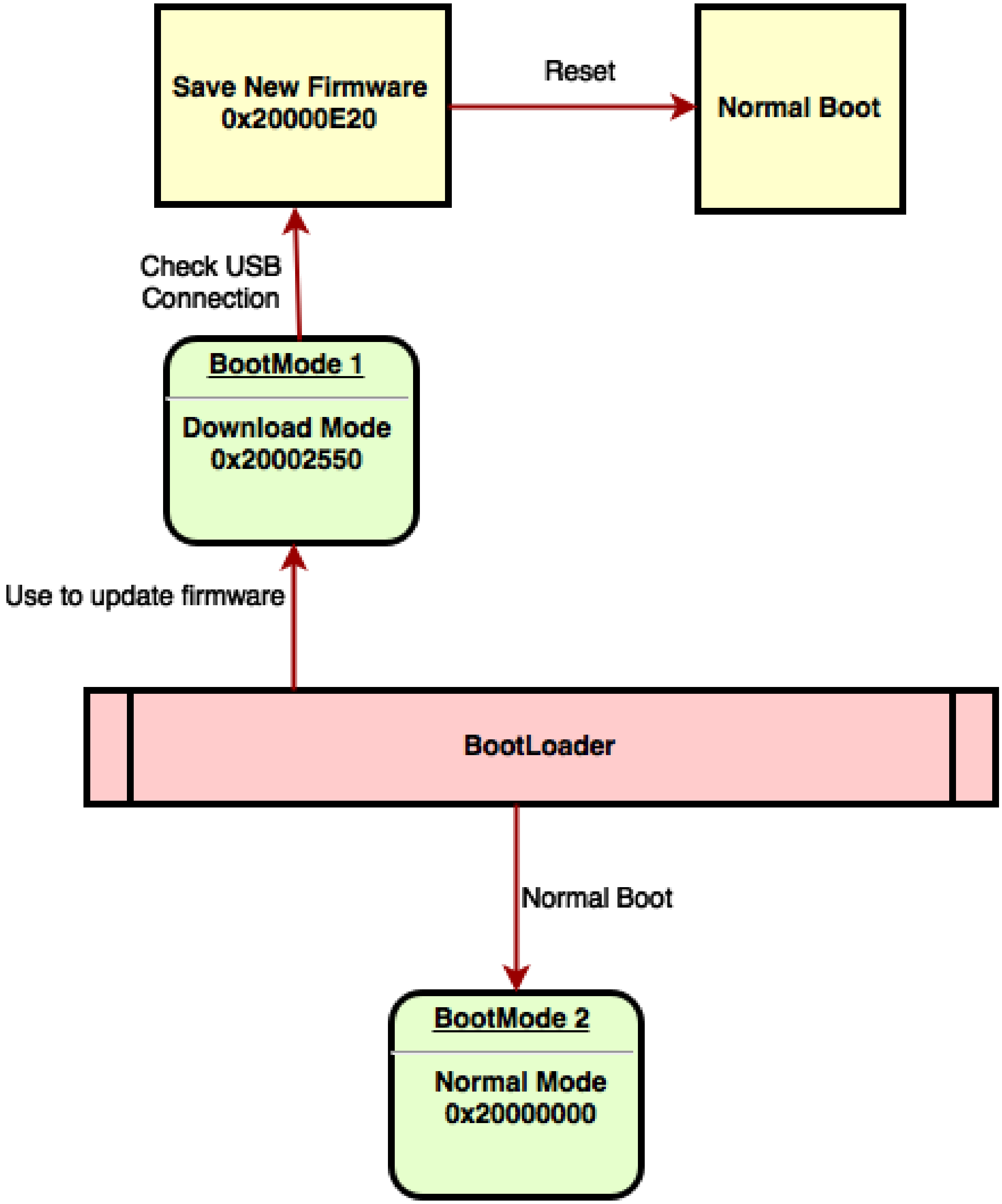


Figure 15: Boot Sequence

5. Memory Map:

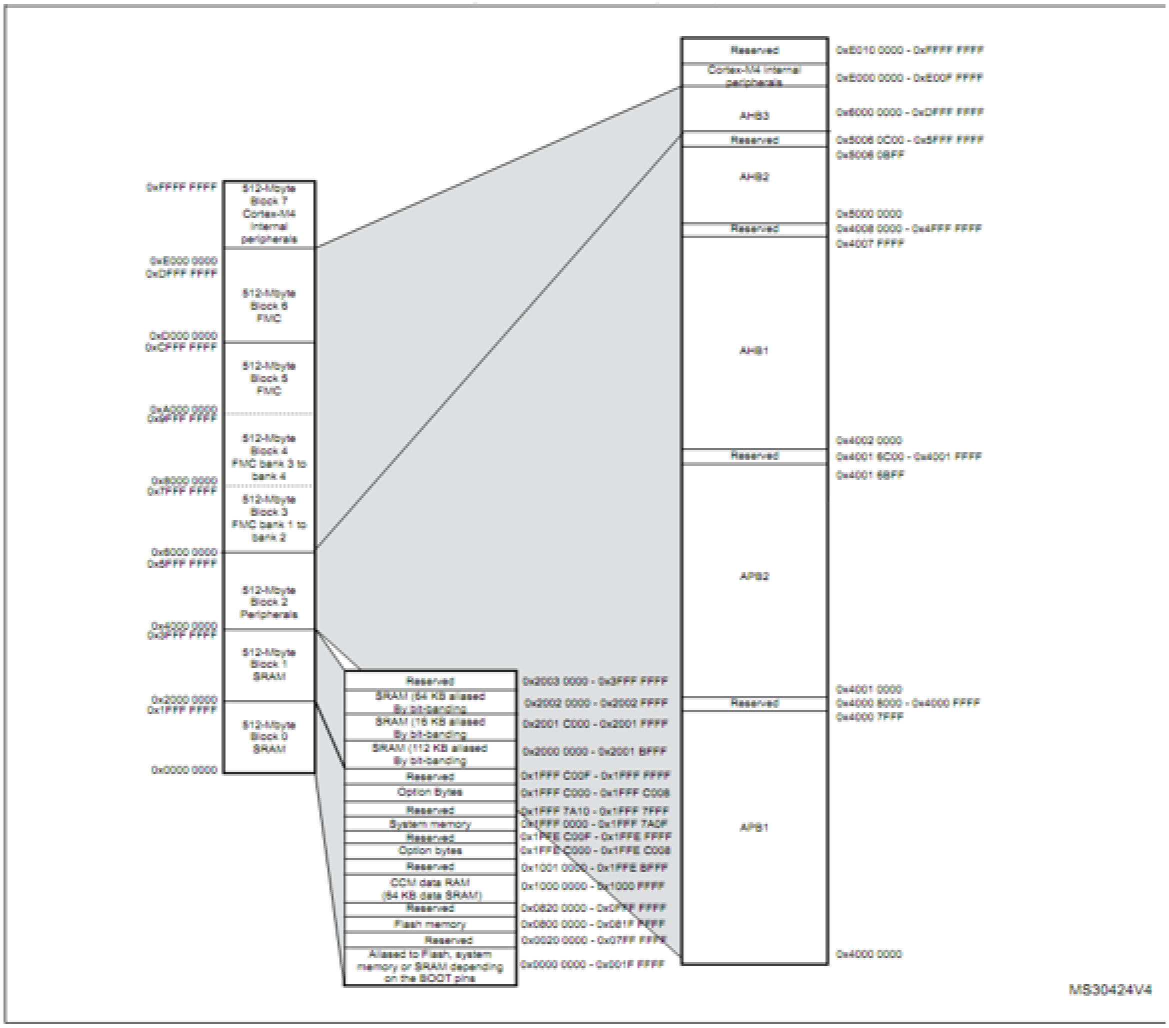

0xFFFF FFFF
512-Mbyte Block 7 Cortex-M4 Internal peripherals
0xE000 0000
0xDFFF FFFF
512-Mbyte Block 6 FMC
0xD000 0000
0xCFFF FFFF
512-Mbyte Block 5 FMC
0xA000 0000
0x9FFF FFFF
512-Mbyte Block 4 FMC bank 3 to bank 4
0x8000 0000
0x7FFF FFFF
512-Mbyte Block 3 FMC bank 1 to bank 2
0x6000 0000
0x5FFF FFFF
512-Mbyte Block 2 Peripherals
0x4000 0000
0x3FFF FFFF
512-Mbyte Block 1 SRAM
0x2000 0000
0x1FFF FFFF
512-Mbyte Block 0 SRAM
0x0000 0000
Reserved
0x2003 0000 - 0x3FFF FFFF
SRAM (64 KB aliased By bit-banding
0x2002 0000 - 0x2002 FFFF
SRAM (16 KB aliased By bit-banding
0x2001 C000 - 0x2001 FFFF
SRAM (112 KB aliased By bit-banding
0x2000 0000 - 0x2001 BFFF
Reserved
0x1FFF C00F - 0x1FFF FFFF
Option Bytes
0x1FFF C000 - 0x1FFF C008
Reserved
0x1FFF 7A10 - 0x1FFF 7FFF
System memory
0x1FFF 0000 - 0x1FFF 7A0F
Reserved
0x1FFE C00F - 0x1FFE FFFF
Option bytes
0x1FFE C000 - 0x1FFE C008
Reserved
0x1001 0000 - 0x1FFE BFFF
CCM data RAM (64 KB data SRAM)
0x1000 0000 - 0x1000 FFFF
Reserved
0x0820 0000 - 0x0FFF FFFF
Flash memory
0x0800 0000 - 0x081F FFFF
Reserved
0x0020 0000 - 0x07FF FFFF
Aliased to Flash, system memory or SRAM depending on the BOOT pins
0x0000 0000 - 0x001F FFFF
Reserved
0xE010 0000 - 0xFFFF FFFF
Cortex-M4 internal peripherals
0xE000 0000 - 0xE00F FFFF
AHB3
0x6000 0000 - 0xDFFF FFFF
Reserved
0x5006 0C00 - 0x5FFF FFFF
0x5006 0BFF
AHB2
0x5000 0000
Reserved
0x4008 0000 - 0x4FFF FFFF
0x4007 FFFF
AHB1
0x4002 0000
Reserved
0x4001 6C00 - 0x4001 FFFF
0x4001 6BFF
APB2
0x4001 0000
Reserved
0x4000 8000 - 0x4000 FFFF
0x4000 7FFF
APB1
0x4000 0000
MS30424V4

Figure 16:Memory Map [14]

Memory graph after boot:

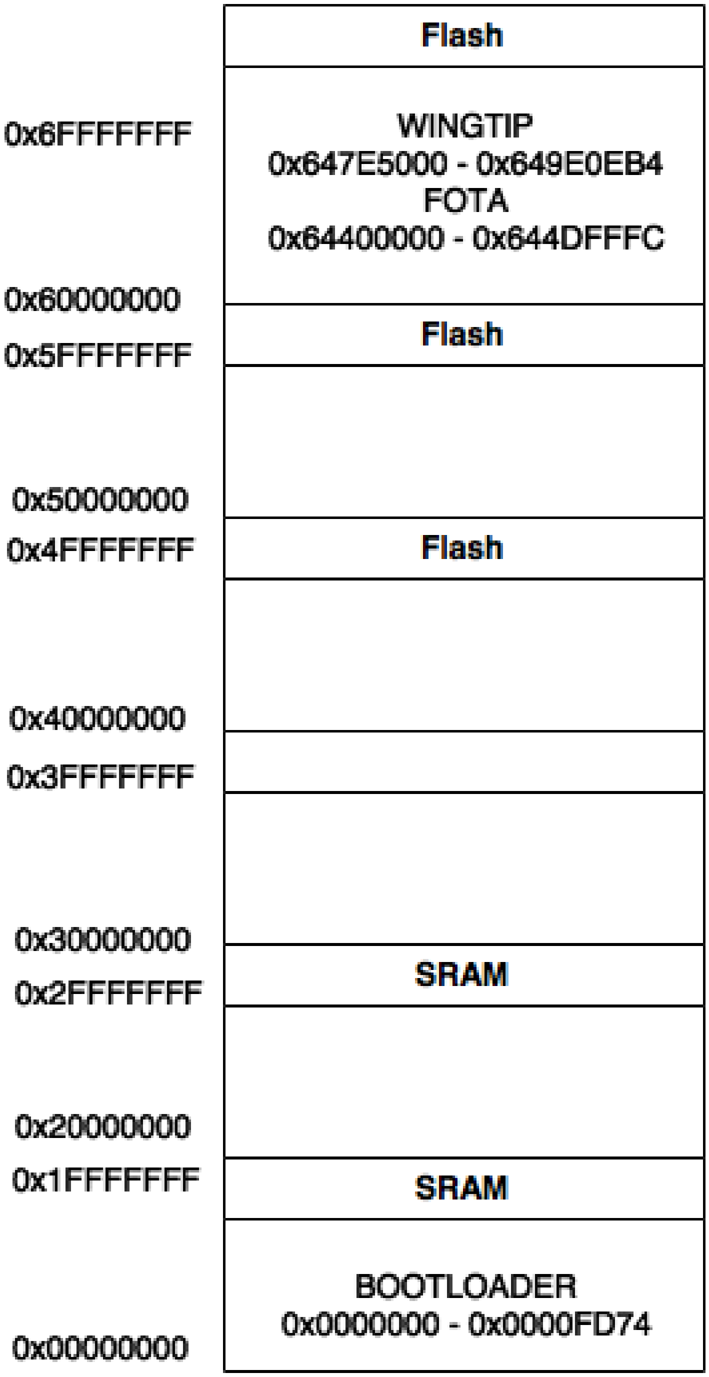
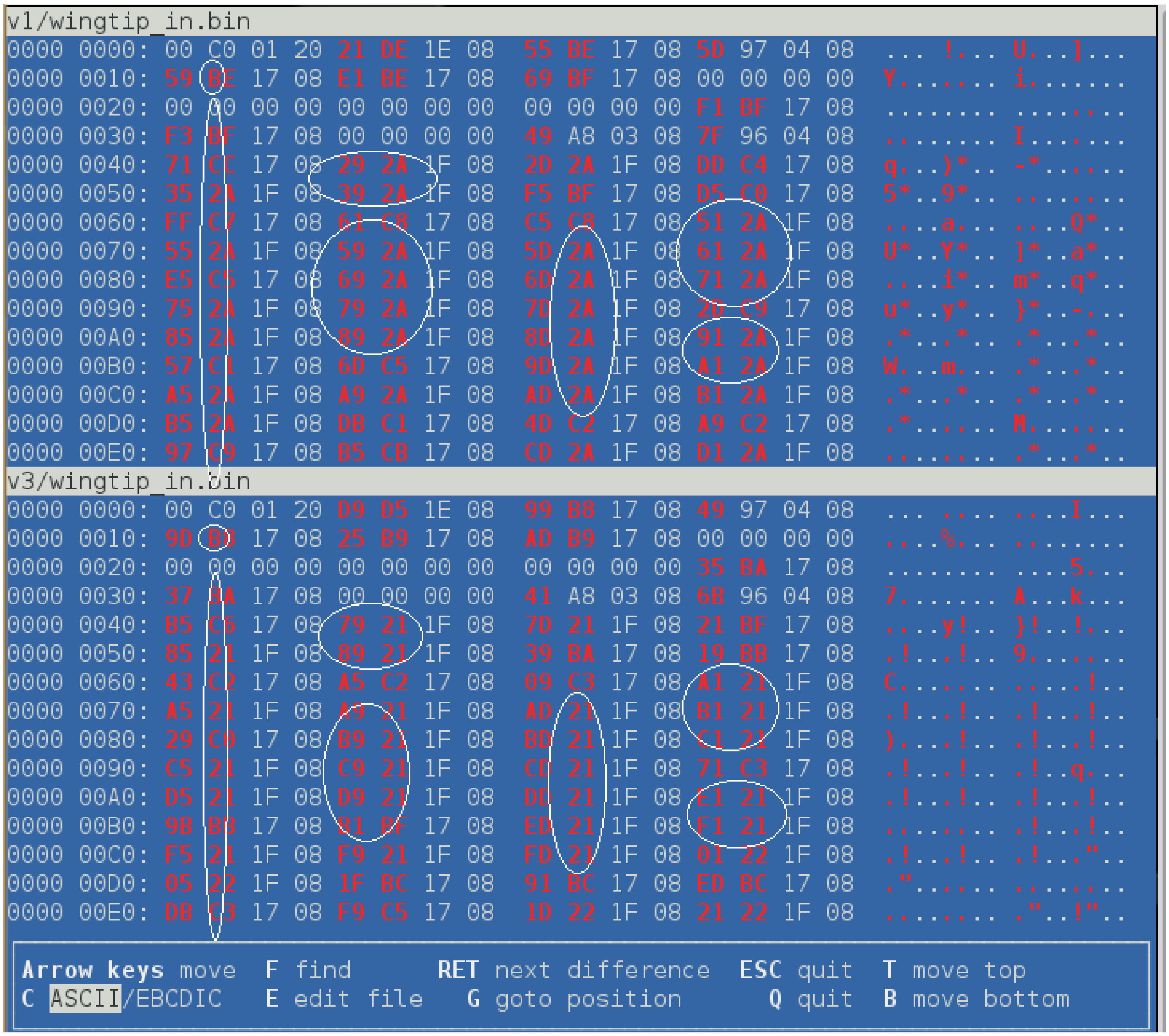


Figure 17: Memory After Boot

6. Reverse Header:

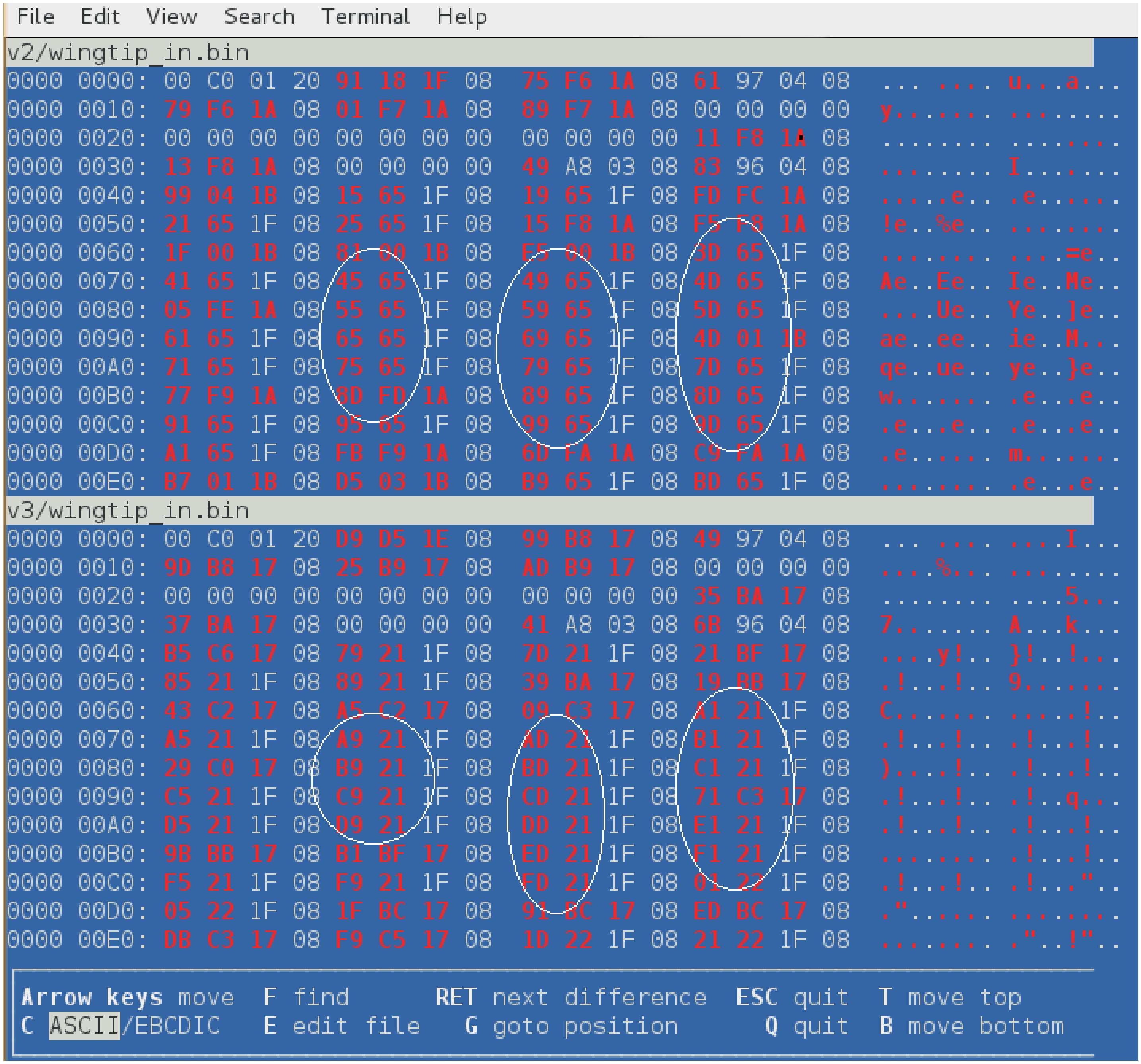
6.1. Compare header of the update firmware

- From the internet, we get 3 different versions of the gearfit's update firmware.
- We used the vfbindiff tool to compare the header of the updates firmware and find some repeat partion.



```
v1/wingtip_in.bin
0000 0000: 00 C0 01 20 21 DE 1E 08  55 BE 17 08 5D 97 04 08  ... !... U...]...
0000 0010: 59 BE 17 08 E1 BE 17 08  69 BF 17 08 00 00 00 00  Y....... i.......
0000 0020: 00 00 00 00 00 00 00 00  00 00 00 00 F1 BF 17 08  ........ ........
0000 0030: F3 BF 17 08 00 00 00 00  49 A8 03 08 7F 96 04 08  ........ I.......
0000 0040: 71 CC 17 08 29 2A 1F 08  2D 2A 1F 08 DD C4 17 08  q...)*.. -*......
0000 0050: 35 2A 1F 08 39 2A 1F 08  F5 BF 17 08 D5 C0 17 08  5*..9*.. ........
0000 0060: FF C7 17 08 61 C8 17 08  C5 C8 17 08 51 2A 1F 08  ....a... ....Q*..
0000 0070: 55 2A 1F 08 59 2A 1F 08  5D 2A 1F 08 61 2A 1F 08  U*..Y*.. ]*..a*..
0000 0080: E5 C5 17 08 69 2A 1F 08  6D 2A 1F 08 71 2A 1F 08  ....i*.. m*..q*..
0000 0090: 75 2A 1F 08 79 2A 1F 08  7D 2A 1F 08 2D C9 17 08  u*..y*.. }*..-...
0000 00A0: 85 2A 1F 08 89 2A 1F 08  8D 2A 1F 08 91 2A 1F 08  .*...*.. .*...*..
0000 00B0: 57 C1 17 08 6D C5 17 08  9D 2A 1F 08 A1 2A 1F 08  W...m... .*...*..
0000 00C0: A5 2A 1F 08 A9 2A 1F 08  AD 2A 1F 08 B1 2A 1F 08  .*...*.. .*...*..
0000 00D0: B5 2A 1F 08 DB C1 17 08  4D C2 17 08 A9 C2 17 08  .*...... M.......
0000 00E0: 97 C9 17 08 B5 CB 17 08  CD 2A 1F 08 D1 2A 1F 08  ........ .*...*..
v3/wingtip_in.bin
0000 0000: 00 C0 01 20 D9 D5 1E 08  99 B8 17 08 49 97 04 08  ... .... ....I...
0000 0010: 9D B8 17 08 25 B9 17 08  AD B9 17 08 00 00 00 00  ....%... ........
0000 0020: 00 00 00 00 00 00 00 00  00 00 00 00 35 BA 17 08  ........ ....5...
0000 0030: 37 BA 17 08 00 00 00 00  41 A8 03 08 6B 96 04 08  7....... A...k...
0000 0040: B5 C6 17 08 79 21 1F 08  7D 21 1F 08 21 BF 17 08  ....y!.. }!..!...
0000 0050: 85 21 1F 08 89 21 1F 08  39 BA 17 08 19 BB 17 08  .!...!.. 9.......
0000 0060: 43 C2 17 08 A5 C2 17 08  09 C3 17 08 A1 21 1F 08  C....... .....!..
0000 0070: A5 21 1F 08 A9 21 1F 08  AD 21 1F 08 B1 21 1F 08  .!...!.. .!...!..
0000 0080: 29 C0 17 08 B9 21 1F 08  BD 21 1F 08 C1 21 1F 08  )....!.. .!...!..
0000 0090: C5 21 1F 08 C9 21 1F 08  CD 21 1F 08 71 C3 17 08  .!...!.. .!..q...
0000 00A0: D5 21 1F 08 D9 21 1F 08  DD 21 1F 08 E1 21 1F 08  .!...!.. .!...!..
0000 00B0: 9B BB 17 08 B1 BF 17 08  ED 21 1F 08 F1 21 1F 08  ........ .!...!..
0000 00C0: F5 21 1F 08 F9 21 1F 08  FD 21 1F 08 01 22 1F 08  .!...!.. .!..."..
0000 00D0: 05 22 1F 08 1F BC 17 08  91 BC 17 08 ED BC 17 08  ."...... ........
0000 00E0: DB C3 17 08 F9 C5 17 08  1D 22 1F 08 21 22 1F 08  ........ ."..!"..

 Arrow keys move   F find       RET next difference  ESC quit   T move top
 C ASCII/EBCDIC    E edit file    G goto position      Q quit   B move bottom
```

Figure 18: Header Firmware 1 and 2



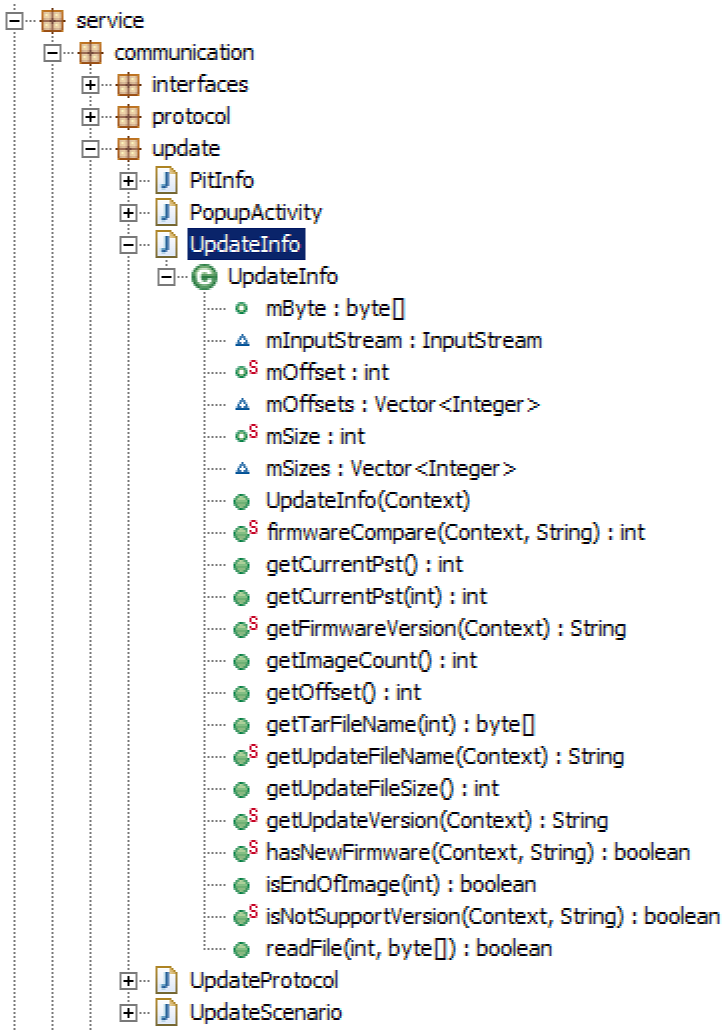
```
v2/wingtip_in.bin
0000 0000: 00 C0 01 20 91 18 1F 08  75 F6 1A 08 61 97 04 08  ... .... u...a...
0000 0010: 79 F6 1A 08 01 F7 1A 08  89 F7 1A 08 00 00 00 00  y....... ........
0000 0020: 00 00 00 00 00 00 00 00  00 00 00 00 11 F8 1A 08  ........ ........
0000 0030: 13 F8 1A 08 00 00 00 00  49 A8 03 08 83 96 04 08  ........ I.......
0000 0040: 99 04 1B 08 15 65 1F 08  19 65 1F 08 FD FC 1A 08  .....e.. .e......
0000 0050: 21 65 1F 08 25 65 1F 08  15 F8 1A 08 F5 F8 1A 08  !e..%e.. ........
0000 0060: 1F 00 1B 08 81 00 1B 08  E5 00 1B 08 3D 65 1F 08  ........ ....=e..
0000 0070: 41 65 1F 08 45 65 1F 08  49 65 1F 08 4D 65 1F 08  Ae..Ee.. Ie..Me..
0000 0080: 05 FE 1A 08 55 65 1F 08  59 65 1F 08 5D 65 1F 08  ....Ue.. Ye..]e..
0000 0090: 61 65 1F 08 65 65 1F 08  69 65 1F 08 4D 01 1B 08  ae..ee.. ie..M...
0000 00A0: 71 65 1F 08 75 65 1F 08  79 65 1F 08 7D 65 1F 08  qe..ue.. ye..}e..
0000 00B0: 77 F9 1A 08 8D FD 1A 08  89 65 1F 08 8D 65 1F 08  w....... .e...e..
0000 00C0: 91 65 1F 08 95 65 1F 08  99 65 1F 08 9D 65 1F 08  .e...e.. .e...e..
0000 00D0: A1 65 1F 08 FB F9 1A 08  6D FA 1A 08 C9 FA 1A 08  .e...... m.......
0000 00E0: B7 01 1B 08 D5 03 1B 08  B9 65 1F 08 BD 65 1F 08  ........ .e...e..
v3/wingtip_in.bin
0000 0000: 00 C0 01 20 D9 D5 1E 08  99 B8 17 08 49 97 04 08  ... .... ....I...
0000 0010: 9D B8 17 08 25 B9 17 08  AD B9 17 08 00 00 00 00  ....%... ........
0000 0020: 00 00 00 00 00 00 00 00  00 00 00 00 35 BA 17 08  ........ ....5...
0000 0030: 37 BA 17 08 00 00 00 00  41 A8 03 08 6B 96 04 08  7....... A...k...
0000 0040: B5 C6 17 08 79 21 1F 08  7D 21 1F 08 21 BF 17 08  ....y!.. }!..!...
0000 0050: 85 21 1F 08 89 21 1F 08  39 BA 17 08 19 BB 17 08  .!...!.. 9.......
0000 0060: 43 C2 17 08 A5 C2 17 08  09 C3 17 08 A1 21 1F 08  C....... .....!..
0000 0070: A5 21 1F 08 A9 21 1F 08  AD 21 1F 08 B1 21 1F 08  .!...!.. .!...!..
0000 0080: 29 C0 17 08 B9 21 1F 08  BD 21 1F 08 C1 21 1F 08  )....!.. .!...!..
0000 0090: C5 21 1F 08 C9 21 1F 08  CD 21 1F 08 71 C3 17 08  .!...!.. .!..q...
0000 00A0: D5 21 1F 08 D9 21 1F 08  DD 21 1F 08 E1 21 1F 08  .!...!.. .!...!..
0000 00B0: 9B BB 17 08 B1 BF 17 08  ED 21 1F 08 F1 21 1F 08  ........ .!...!..
0000 00C0: F5 21 1F 08 F9 21 1F 08  FD 21 1F 08 01 22 1F 08  .!...!.. .!..."..
0000 00D0: 05 22 1F 08 1F BC 17 08  91 BC 17 08 ED BC 17 08  ."...... ........
0000 00E0: DB C3 17 08 F9 C5 17 08  1D 22 1F 08 21 22 1F 08  ........ ."..!"..

 Arrow keys move    F find        RET next difference   ESC quit   T move top
 C ASCII/EBCDIC     E edit file     G goto position       Q quit   B move bottom
```

Figure 19: Header Firmware 2-3

IV. Vulnerability point:

1. Modification Attack:

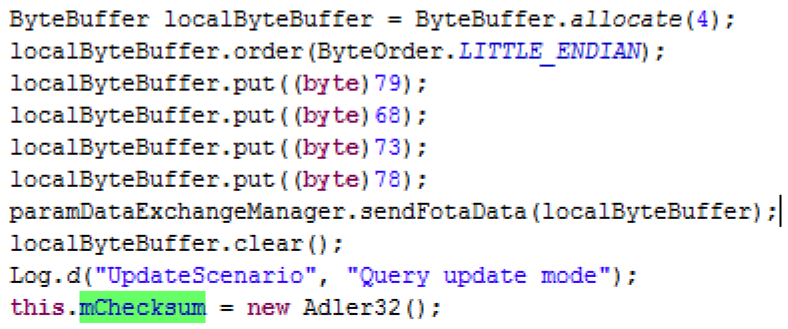
Using some technique to reverse structure of apk file. We discover a strange directory named “firmware” which contained a bin file using for update smartwatch firmware directly. Regarding java code, we found some class link to this file.



```
service
  communication
    interfaces
    protocol
    update
      PitInfo
      PopupActivity
      UpdateInfo
        UpdateInfo
          mByte : byte[]
          mInputStream : InputStream
          mOffset : int
          mOffsets : Vector<Integer>
          mSize : int
          mSizes : Vector<Integer>
          UpdateInfo(Context)
          firmwareCompare(Context, String) : int
          getCurrentPst() : int
          getCurrentPst(int) : int
          getFirmwareVersion(Context) : String
          getImageCount() : int
          getOffset() : int
          getTarFileName(int) : byte[]
          getUpdateFileName(Context) : String
          getUpdateFileSize() : int
          getUpdateVersion(Context) : String
          hasNewFirmware(Context, String) : boolean
          isEndOfImage(int) : boolean
          isNotSupportVersion(Context, String) : boolean
          readFile(int, byte[]) : boolean
      UpdateProtocol
      UpdateScenario
```

Figure 20: Update Class Structure

From this evidence, we conclude that this is file used to update firmware. Subsequently, we continue finding GearFit Manager whether using checksum function or not.



```
ByteBuffer localByteBuffer = ByteBuffer.allocate(4);
localByteBuffer.order(ByteOrder.LITTLE_ENDIAN);
localByteBuffer.put((byte)79);
localByteBuffer.put((byte)68);
localByteBuffer.put((byte)73);
localByteBuffer.put((byte)78);
paramDataExchangeManager.sendFotaData(localByteBuffer);
localByteBuffer.clear();
Log.d("UpdateScenario", "Query update mode");
this.mChecksum = new Adler32();
```

Figure 21: CheckSum Function

GearFit Manager's checksum is Adler32 and it implement normal standard support in Android. SamSung create Adler32 checksum value on buffer byte after reading from update firmware file that mean it only ensure integrity while transferring from smartphone to smartwatch. Therefore, we can modify firmware file without concerning about checksum value. With Virtuous Ten Studio tool, we rebuild GearFit Manager apk file with modified firmware and update modified firmware to GearFit Watch.

2. <u>Session Hijacking:</u>

2.1. <u>Over view:</u>

The classes in Scup library use to connect to watch apps are private. So the idea is that we will implement a new public class to create a new illegal connection to other watch apps. Scup uses ScupCommunicator class to initiate connection to watch and connection services are only using the package as constructor parameter. We can compromise application by change the target connection.

2.2. <u>Reverse ScupComminication class:</u>

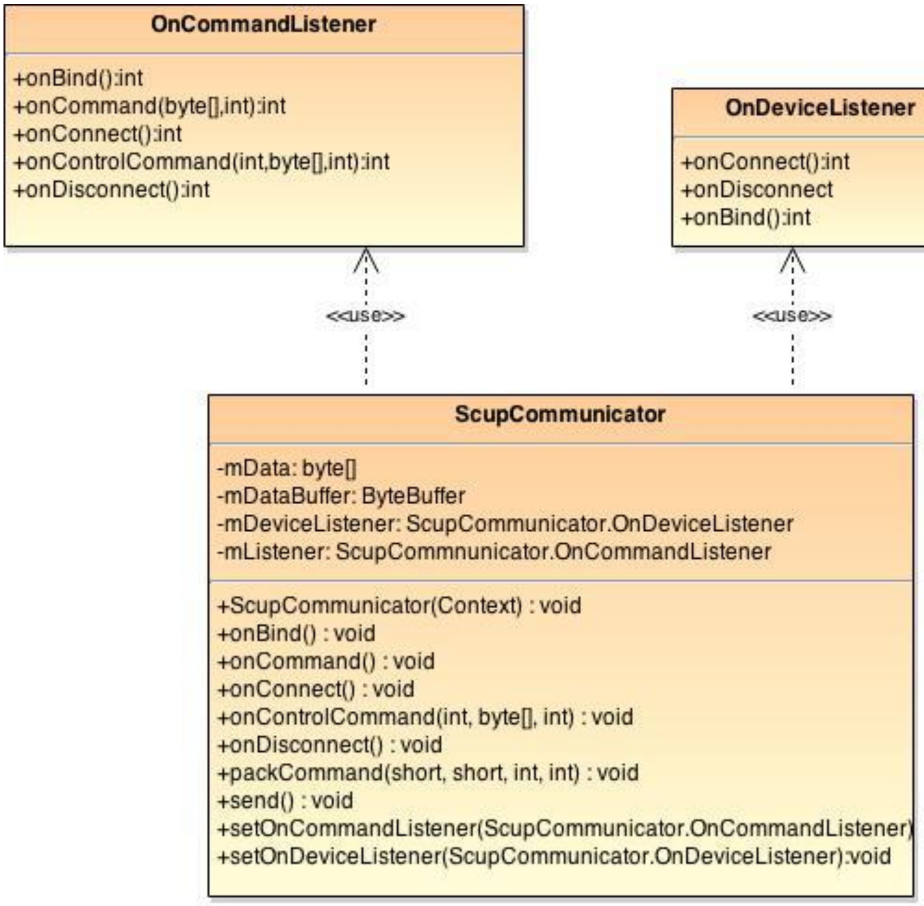

OnCommandListener
+onBind():int
+onCommand(byte[],int):int
+onConnect():int
+onControlCommand(int,byte[],int):int
+onDisconnect():int
OnDeviceListener
+onConnect():int
+onDisconnect
+onBind():int
<<use>>
<<use>>
ScupCommunicator
-mData: byte[]
-mDataBuffer: ByteBuffer
-mDeviceListener: ScupCommunicator.OnDeviceListener
-mListener: ScupCommnunicator.OnCommandListener
+ScupCommunicator(Context) : void
+onBind() : void
+onCommand() : void
+onConnect() : void
+onControlCommand(int, byte[], int) : void
+onDisconnect() : void
+packCommand(short, short, int, int) : void
+send() : void
+setOnCommandListener(ScupCommunicator.OnCommandListener)
+setOnDeviceListener(ScupCommunicator.OnDeviceListener):void

Figure 22: ScupCommunicator Class Diagram

2.3. **Our Approach:**

The weakness point is that watch does not have any authenticated method when application create service connection to connect with watch application. The parameters are only package name and Samsung service name. After host application connect successful to watch app, we create one more new connection to the other application and send the command. Watch app receive the command and perform because it believes that command come from legal application.

Unfortunately, all of connection function in Scup library is private function. We need to implement a new function by integrating from these functions and build a new library.

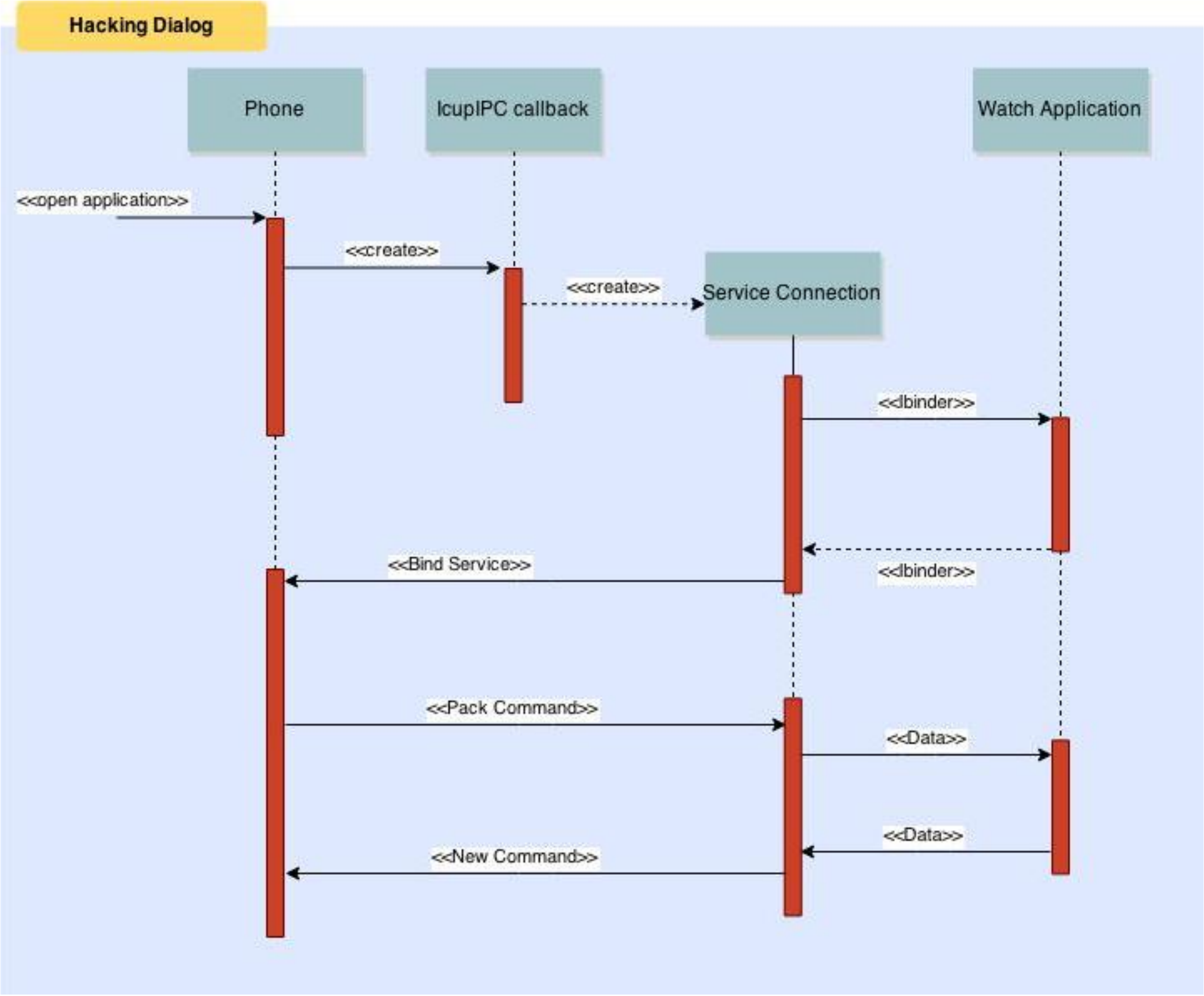


Figure 23: Compromise process

3. **Future work:**

Base on result of modification attack, we will continue perform many attacks to smartwatch's authentication mechanism:

- Disable authentication connection with smartphone.
- Add user's private information (Location, heath, phone message….) into response message to smartphone.

Regarding Session Hijacking, from this success, we will improve to hijack real-time connection between phone and smartwatch by a hidden service running on smartphone and retrieve important information from network flow.

References

[1] Bluetooth specification v4.0 [Online]

https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=229737

[2] Security, Bluetooth Smart (Low Energy) [Online]

https://developer.bluetooth.org/TechnologyOverview/Pages/LE-Security.aspx

[3] Bluetooth Low Energy SMP Pairing [Online]

https://community.freescale.com/thread/332191

[4] Project Ubertooth [Online]

https://github.com/greatscottgadgets/ubertooth

[5] How to build Ubertooth One [Online]

https://github.com/greatscottgadgets/ubertooth/wiki/Build-Guide

[6] Bluetooth: With Low Energy Comes Low Security [Online]

https://www.usenix.org/conference/woot13/workshop-program/presentation/ryan

[7] Motivating the Problem [Online]

http://ubertooth.blogspot.fr/2013/02/motivating-problem.html

[8] BlueSniff: Eve meets Alice and Bluetooth [Online]

http://static.usenix.org/event/woot07/tech/full_papers/spill/spill_html/

[9] Naresh Gupta, Inside Bluetooth Low Energy, Artech House (March 1, 2013), page 197

[10] Bluetooth programming with Python - PyBluez [Online]

http://people.csail.mit.edu/albert/bluez-intro/c212.html

[11] Texas Instrument CC2400 [Online]

http://www.ti.com/product/cc2400

[12] Bluetooth Developer Portal [Online]

https://developer.bluetooth.org/TechnologyOverview/Pages/BLE.aspx

[13] Bluetooth_Low_Energy_CSR.pdf [Online]

https://www.bluetooth.org/DocMan/handlers/DownloadDoc.ashx?doc_id=227336

[14] Technical Reference ManualARM® Cortex®-M4 Processor, Revision: r0p1, 2013

[15] STM32F3 and STM32F4 Series Cortex -M4 programming manual

[16] SamSung, CUP Programming Guide

[17] The - Eagle, Chris, IDA Pro Book The Unofficial Guide to the World's Most Popular Disassembler, 2nd New edition (30 May 2011)

[18] Zachry H. Basnight, First Lieutenant, Firmware counterfeiting and modification attacks on programmable logic controllers thesis, March 2013

[19] Andrea Barisani, Daniele Bianco, Practical Exploitation of Embedded Systems, HITBSecConf 2012

[20] SamSung GearFit Produce [online]

http://www.samsung.com/us/mobile/wearable-tech/SM-R3500ZKAXAR

[21] SamSung GearFit on XDA Forum [Online]

http://forum.xda-developers.com/gear-fit