ANALYSING AND ASSESSING SECURITY OF FITBIT CHARGE HR

Kou Yong Kang[1]. Koh Ming Yang[2].
[1]Dunman High School. 10 Tanjong Rhu Rd, Singapore 436895
[2]DSO National Laboratories. 20 Science Park Dr, Singapore 118230

ABSTRACT

With the advent of wearables, technology has become increasingly close to us, and at the same time, collecting a lot more data from us than we could ever imagine. We decided to look into Fitbit Charge HR, a fitness tracker produced by Fitbit Inc., to investigate its privacy capabilities. We will present the communication protocol used between the tracker and smartphone, along with observations made throughout this investigation.

INTRODUCTION

What is an embedded device?

An embedded device is an object that contains a special-purpose computing system. Embedded systems have extensive applications in consumer, commercial, automotive, industrial and healthcare markets. Embedded devices usually have limited computing resources and strict power requirements[1]. Some examples include ATM, smartphones, fitness trackers and pacemakers etc. Wearables are also an emerging form of embedded devices.

Why choose wearables?

With the advent of wearables, technology has become increasingly closer to us, and at the same time, collecting a lot more data from us than we could ever imagine. There is a plethora of wearable devices collecting sensitive information from us, with increasing precision and granularity. As wearable technology continues to become prevalent, its application will only become more widespread, with more sensors embedded around us and possibly inside us. Smart blood glucose monitors and smart pacemakers are just a few of these possibilities. If any of these devices are ever compromised, there will definitely be huge implications concerning more than just privacy issues, exposing ourselves to threats that may even put our lives at stake.

Wearables are designed to be donned at all times and therefore has a need to keep its power usage at its minimum. This design consideration influenced many of these wearables to adopt Bluetooth Low Energy (BLE) as their mode of transmission. The BLE standard is designed with security and privacy considerations, but with many of these features made optional, several of these key features not being utilised.

Fitbit Charge HR is chosen to be analysed due to Fitbit's high market share[2] in the fitness tracker market and also due to the fact that it features an optical heart rate sensor, a feature that is becoming integral to both fitness trackers and also smartwatches. More importantly, there had been several instances of security vulnerabilities in Fitbit products, such as the

[1] http://whatis.techtarget.com/definition/embedded-device
[2] http://www.wareable.com/fitbit/fitness-tracker-sales-2015-fitbit-1169

disclosure of sensitive user data[3] and also security issues [1] and exploits[4] with Fitbit devices.

Aim of the project

The main aim of this project is to take a look at Fitbit Charge HR to see if vulnerabilities discovered previously still exist on current devices. The project will help give insights into the synchronisation protocol along with the encryption techniques employed to see if data can be extracted in one way or another. A demonstration will also be performed to show how sensitive information can be stolen by malicious hackers.

MATERIALS AND METHODS

Past efforts

There has been a previous attempt [2] which looked at Fitbit Flex, an older model that predates the Charge HR. The analysis looked at several aspects of Fitbit Flex, from Bluetooth to the Android application and also the server communication. This allowed us to gain initial understanding into how Fitbit designed their protocols but did not give much details into the synchronisation mechanism and encryption. Also, Fitbit has since updated the firmware for various devices and the smartphone applications.

Tools

In order to analyse the underlying protocol in Fitbit devices, we analysed the Fitbit Android application. Decompiling the .apk file using dex2jar[5], we were able to get back some of the java source code when viewing it using JD-GUI[6]. The Android SDK[7] was also used for logging purposes to better understand the obfuscated source code while performing static analysis.

The Fitbit Android application was ran on a Samsung Galaxy S3 running Android 4.4.4. Other than running the Fitbit mobile applications, another application used to retrieve data from Fitbit Charge HR was nRF Master Control Panel (BLE)[8], which allowed for standard BLE communications with the Fitbit Charge HR without the need for the Fitbit application.

An attempt was also made to sniff the BLE traffic between the smartphone and Fitbit Charge HR using CC2540 BLE development platform made by Texas Instrument. Texas Instrument also provided a software, SmartRF Protocol Packet Sniffer[9] to be used in conjunction with CC2540 after it is programmed to be in sniffing mode.

FITBIT APPLICATION ANALYSIS

The Fitbit mobile application is the only method to synchronise aggregated data from Fitbit devices to the Fitbit servers. It is therefore vital to take a deeper look into the Fitbit application to better understand the underlying protocols and mechanisms.

[3] http://techcrunch.com/2011/07/03/sexual-activity-tracked-by-fitbit-shows-up-in-google-search-results/
[4] http://www.theregister.co.uk/2015/10/21/fitbit_hack
[5] http://sourceforge.net/projects/dex2jar/
[6] http:// jd.benow.ca/
[7] http://developer.android.com/sdk/index.html
[8] http://play.google.com/store/apps/details?id=no.nordicsemi.android.mcp
[9] http:// www.ti.com/tool/packet-sniffer

Synchronisation Process

Figure 1: Overview of Fitbit Synchronisation

Synchronisation of Fitbit devices are performed over BLE via either the USB Sync dongle or smartphone, with data directly forwarded to Fitbit servers over an encrypted connection. Fitbit servers will send a response back to the USB Sync dongle or smartphone, which will then forward the response to the Fitbit device, as show in Figure 1.

The Fitbit USB Sync dongle came along with the device, and is observed to operate on a proprietary protocol, designed to interface only with the official Fitbit Connect[10] application available for Windows and Mac OS X. An open-source alternative, Galileo[11], is a Python script which achieves the same task. From the data dump saved by Galileo, which dumps the data sent to and fro Fitbit servers, no obvious patterns could be observed, implying that the data is encrypted. Also, Fitbit account credentials is not required in any of these cases.

When syncing using a smartphone, the Fitbit mobile application is necessary for pairing and to perform synchronisation. Data remains encrypted when transiting through the smartphone. This is evident as syncing cannot be performed when offline and the dashboard will not have updated data until syncing with Fitbit servers is complete. When offline, the dashboard can be refreshed with live data but this is not cached on the smartphone, neither can data with more granularity be retrieved this way. When no Fitbit devices are around, data showing the statistics as of the last synchronisation session will be fetched from the Fitbit servers, in a similar manner to how data is displayed on the smartphone after synchronisation.

Decompilation and Logging

This analysis is done using a Samsung Galaxy S3 and a set of tools running on Ubuntu.

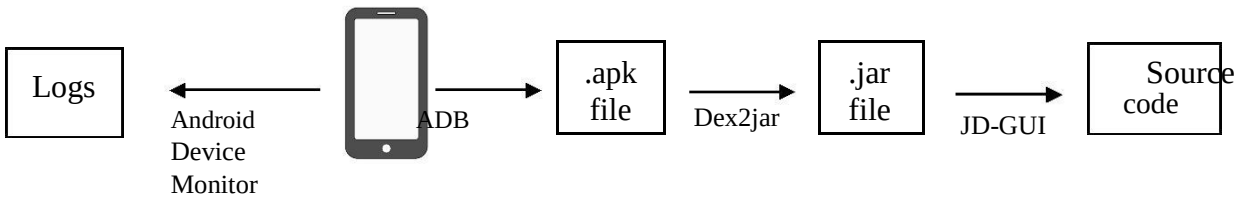

Figure 2: Method used to view Fitbit application source code and behaviour

[10] http://www.fitbit.com/sg/setup
[11] https://bitbucket.org/benallard/galileo

The Fitbit Android application is extracted from the smartphone using adb from Android SDK. The resulting application file (.apk file) is a zipped and signed archive of the application. The archive contains the machine code of the Fitbit application, which is difficult to understand.

In order to get source code that can be more easily understood and analysed, the application was decompiled using dex2jar into its java source. The resulting java source (.jar file) was opened using JD-GUI to display all the decompiled classes and methods. It is noted that the source code has been obfuscated. To better understand the obfuscated code, we made use of Android Device Monitor (also part of Android SDK) to look at the logs generated while running the application.

ANALYSIS & DISCUSSION

Live Data Mode

Upon a detailed inspection of the source code, an interesting finding is the "live data" class. Although the methods under the "live data" class did not reveal anything particularly useful, there is brief mention of variables that are related to the data that Fitbit Charge HR collects, such as steps, calories and heart rate. The "live data" class also led to the discovery of another class named "GalileoOtaMessages.class".

```
    public RFLiveDataPacket(byte[] paramArrayOfByte)
    {
      this.timeStamp = b.b(paramArrayOfByte, 0);
      this.steps = b.b(paramArrayOfByte, 4);
      this.distance = b.b(paramArrayOfByte, 8);
      this.calories = b.c(paramArrayOfByte, 12);
      this.elevation = ((short)(b.c(paramArrayOfByte, 14) / 10)); if
      (paramArrayOfByte.length >= 18)
      {
        this.veryActiveMinutes = b.c(paramArrayOfByte, 16);
        if (paramArrayOfByte.length < 20) {
          break label115;
        }
        this.heartRate = ((short)(paramArrayOfByte[18] & 0xFF));
      }
      for (this.heartRateConfidence = ((short)(paramArrayOfByte[19] & 0xFF));;
this.heartRateConfidence = 0)
      {
        return;
        this.veryActiveMinutes =
        0; break;
        label115:
        this.heartRate =
        0;
      }
    }

  public static int b(byte[] paramArrayOfByte, int paramInt)
  {
    return paramArrayOfByte[(paramInt + 3)] << 24 & 0xFF000000 |
paramArrayOfByte[(paramInt + 2)] << 16 & 0xFF0000 | paramArrayOfByte[(paramInt + 1)]
<< 8 & 0xFF00 | paramArrayOfByte[paramInt] & 0xFF;
  }
```

Figure 3: Snippet of code from "GalileoOtaMessages.class"

Figure 3 shows the code that revealed the message format of live data packets and also how the packets are being read into memory. From the code above, we could deduce that the data is in little endian byte order and that each packet has a length of either 18 or 20 bytes. It is worth noting that live data packets do not appear to be encrypted.

Byte no.	Length (bytes)	Data represented
0-3	4	Timestamp
4-7	4	Steps
8-11	4	Distance
12-13	2	Calories
14-15	2	Elevation
16-17	2	Very active minutes
18	1	Heart rate
19	1	Heart rate confidence

Table 1: Summary of message format for live data packets

The message format of each live data packet is shown in Table 1. Each live data packets are can be either 18 or 20 bytes due to the last 2 bytes being related to heart rate, which will be omitted by Fitbit devices without a heart rate sensor like Fitbit Charge HR.

From the logs, each time that a live data packet comes in, there will be a log statement which contains processed data in the same sequence as the live data packets. It is noted that the timestamp is in the UNIX format, i.e. seconds since 1 Jan 1970 UTC 00:00.

Fitbit Charge Authentication

We also looked into the Bluetooth communication between Fitbit Charge HR and smartphone. Much like what was described in a previous analysis of Fitbit Flex [2], authentication is done by computing a MAC over a random number generated (observed to be 3 digits consistently in dynamic analysis) by Fitbit Charge HR, using a CBC-MAC with the XTEA 64-bit block cipher, outlined in "TrackerAuthUtils.class" The XTEA block cipher is provided by Spongy Castle[12], a repackaged version of Bouncy Castle for Android. The version of Spongy Castle included in the Fitbit application is 1.47.0.2, released in 2012. Other than for authentication, Spongy Castle has no other apparent use in the application.

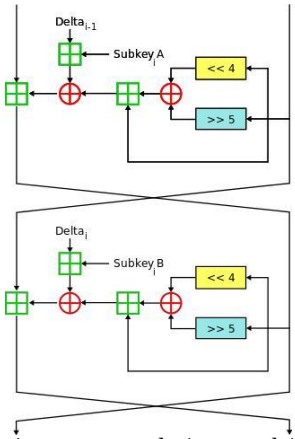

Figure 4: Diagram showing two rounds (one cycle) of the XTEA block cipher
Source: Wikimedia

[12] https://rtyley.github.io/spongycastle/

Fitbit Charge HR Synchronisation

Every time that the Fitbit application is run with a working internet connection, a synchronisation session with Fitbit servers will be triggered. This process essentially captures a megadump (a variable-length byte array containing data to be synced) from Fitbit Charge HR and upload it onto Fitbit servers, then the Fitbit server will then return a megadump to be uploaded onto Fitbit Charge HR. Due to packet size limitation of BLE, all the megadump are broken down into 20 bytes per packet for transfer. The megadump that is pieced together from all the 20-bytes long packets are then verified before upload. This synchronisation with Fitbit servers is performed before the above mentioned authentication occurs. Therefore, it implies that authentication is only necessary for retrieving live data from Fitbit Charge HR.

Fitbit Charge HR Cryptography

A brief analysis of the underlying encryption protocol was also conducted. "SecureDataCoder.class" is the only cryptographic module we found in the application, the java cipher class is imported into this particular class, with AES being used. A message digest is also initialised to use the MD5 algorithm.

Other than identifying the encryption module, Fitbit application log statements also brought to our attention two credential files while seeing the "SecureDataCoder.class" being evoked. They are "authinfo_credentials.json" and "trackerAuthCredentials.json". The two json files are deduced to contain encryption keys used for different purposes. As these two json files are stored in the private data directory (/data/data/com.fitbit.FitbitMobile), direct access to them is restricted by the Android OS. However, as the smartphone used for testing is rooted, it was then possible to extract out these two keys. Below are more details on each of these keys.

<u>authinfo_credentials.json</u>

- length: 128 bytes
- created only after logging in with a Fitbit account
- user will be logged out if the file is deleted, implying that it contains the login credentials
- repeatedly used to initialise instance of "SecureDataCoder.class" as observed from the logs

<u>trackerAuthCredentials.json</u>

- length: 152 bytes
- created just before the first live data session
- retrieved from Fitbit servers, not generated locally on the device
- used to initialise instance of "SecureDataCoder.class" before every MAC authentication challenge
- likely to be crucial for MAC challenge as it is read just before every MAC challenge

By logging the Fitbit application, we have also identified the customised UUID of the Generic Attribute Profile (GATT) services and characteristics, in which data is being transferred between the Fitbit Charge HR and smartphone. Table 2 below is a list of all the UUID with their function known, the standard BLE GATT characteristics are not listed below.

UUID	Function
adabfb01-6e7d-4601-bda2-bffaa68956ba	Characteristic for downloading data from device
adabfb02-6e7d-4601-bda2-bffaa68956ba	Characteristic for uploading data to device
558dfa01-4fa8-4105-9f02-4eaa93e62980	Characteristic for downloading live data

Table 2: List of commonly accessed UUID

Summary

With a basic understanding of the underlying mechanisms of the Fitbit application, we now have a basic idea of how the synchronisation process is being carried out, as shown below.

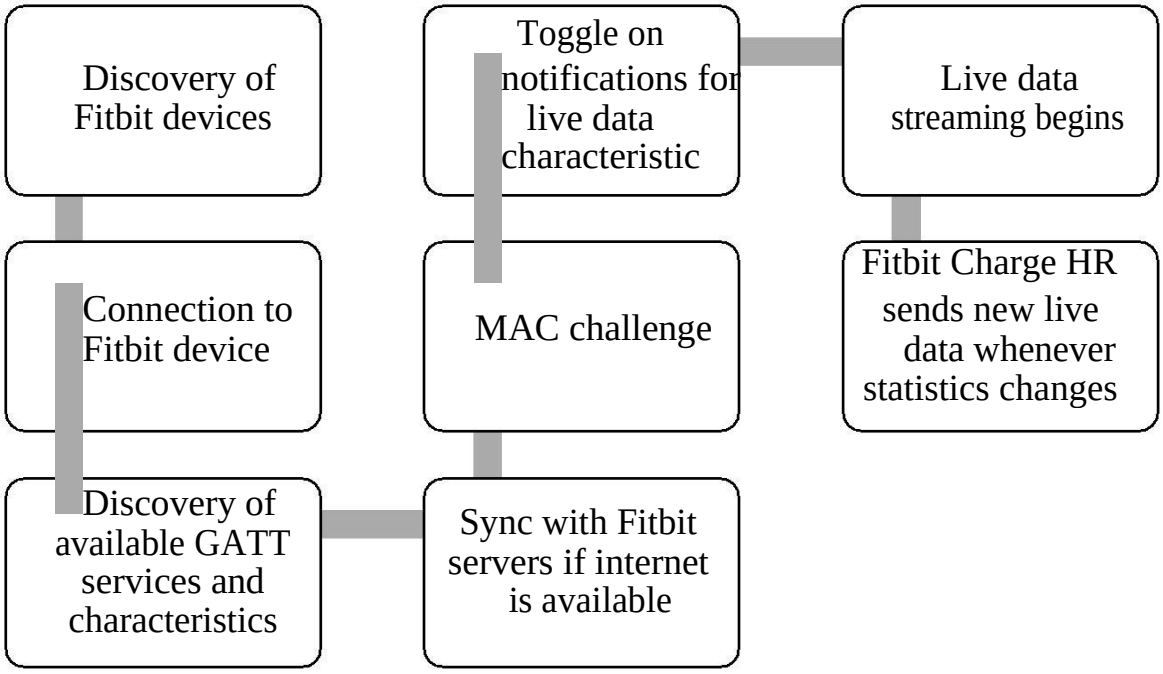

Figure 5: Outline of each sync session

BLE ANALYSIS

The BLE analysis is done via two approaches, firstly by passively sniffing the BLE connection followed by actively connecting to Fitbit Charge HR over BLE.

Passive sniffing

Method

With the format of live data packets known, the best way to find out if they are encrypted is to capture them over the air. To accomplish that, we used a commercially available BLE development board, the Texas Instrument CC2540 USB dongle, as a BLE sniffer. CC2540 works with the SmartRF Protocol Packet Sniffer provided by Texas Instrument. The CC2540 is able to fully capture BLE traffic by following the channel hopping of Fitbit Charge HR and smartphone upon successful capture of the connection packet containing all the connection information. The sniffer programme provided by Texas Instrument is able to parse the relevant headers and output the captured packets in a .psd file, which can be converted into pcap format using open-source tools[13] and viewed in Wireshark[14]. The alternative is to view them directly in the sniffer programme itself.

[13] https://github.com/doggkruse/smartRFtoPcap
[14]https://www.wireshark.org/

Results

All the BLE traffic can be captured by the sniffer beginning from the initial pairing to the live data output. The raw data sent to the Fitbit servers via both the USB Sync dongle and smartphone could be captured but there were no obvious patterns in them, thereby allowing us to confidently infer that all these data are encrypted by Fitbit Charge HR itself. Since there are no clues on how the data is being encrypted, no further attempt was made to decode the data by brute force. The packets captured reaffirmed the observations made in the logs. By looking at the live data pushed to the Fitbit application by Fitbit Charge HR, we are also able to confirm that live data packets are not encrypted. Base on the known packet format mentioned in Section 3.3.1, it was possible to directly parse the raw data by following that format. An example is shown below.

EB	96	82	56	B6	14	00	00	5A	16	3A	00	A2	07	8C	00	13	00	45	02
Timestamp 0x568296EB = 1451398891s ➔ 12/29/2015, 10:21:31 PM				Step count 0x000014B6 = 5302 steps				Distance 0x000014B6 = 3806810m				Calories 0x07A2 = 1954 calories		Floors 0x008C = 140ft ➔ 14 floors		Active minutes 0x0013 = 19 min		Heart rate 0x45 = 69 bpm	Heart rate confidence 0x02 = 2

Table 3: Live data packet decoded example

In one of the sessions, there was a firmware update for Fitbit Charge HR, which is observed to be 298137-bytes long, sent to the device through the Fitbit application.

Active connection

As live connections are not always available for capture, we also tried a more active approach, by directly connecting to Fitbit Charge HR over BLE.

Method

Directly connecting to Fitbit Charge HR over BLE, we can see if any data can be retrieved directly. This was done using nRF Master Control Panel, an Android application designed for BLE developers to scan and communicate with BLE products. Although it is a generic BLE debugging application, it is robust enough to be able to connect to BLE devices and properly perform standard BLE operations and also allow for BLE interactions.

Results

The application could pick up the advertisement packet sent out by Fitbit Charge HR without any problems, offering an option to establish a direct link. Connecting to Fitbit Charge HR will stop it from sending out anymore advertisement packets, and allow for more data to be sent directly to the application. All the GATT services adopted by Bluetooth Special Interest Group (SIG) are parsed correctly and all the characteristics under these services could be read correctly. However, when it comes to reading the live data characteristic (UUID `558dfa01-4fa8-4105-9f02-4eaa93e62980`), the data received is either a null byte, or it will be live data that is not up-to-date and will also not be updated automatically. Reading live data properly is possible if the Fitbit application is already set up on that device and left to run in the background (by activating the call notification feature). This implies that authentication is necessary for live data to be read. Once authenticated, the notification for live data characteristic could be toggled on to allow Fitbit Charge HR to automatically push new live data values to the application. It has been verified that the live data values received by the application matches that of displayed on the LED panel of Fitbit Charge HR.

SUMMARY

In short, the live data packets can be captured in transit as they are unencrypted. However, to be able to retrieve live data directly, one must be authenticated in the first place. Throughout the BLE analysis, one thing to note is that the Bluetooth MAC address of Fitbit Charge HR did not change at all, which making it possible to trace a device and indirectly track the Fitbit device owner. The BLE specifications has a feature that allows for the MAC address to be changed frequently to preserve privacy of users, however, this was observed to not be implemented.

CONCLUSION AND FUTURE WORK

In conclusion, our analysis has revealed the underlying synchronisation mechanism used by the Fitbit Android application to transfer data from Fitbit Charge HR to Fitbit servers, with the smartphone acting as an intermediate that does not process the encrypted data to be synced with Fitbit servers. Our analysis of the application also corroborated with previous analysis [2] that BLE communication between Fitbit Charge HR and smartphone is authenticated with a MAC in CBC mode using XTEA provided by Spongy Castle and we further showed that authentication is not required for syncing with Fitbit servers. The decompiled source code not only gave clues about the ciphers used in the application, it also led to the discovery and extraction of two keys used by the application. Furthermore, the code also revealed the message format of live data and helped to ascertain that the BLE packets captured afterwards are indeed live data. Inspecting the logs generated by the Fitbit application also revealed the nature of unknown GATT services and characteristics. The synchronisation process becomes much clearer with all these insights.

Further analysis done by sniffing the BLE communication between Fitbit Charge HR and smartphone revealed several key points. First of all, data that is supposed to be sent to Fitbit servers are indeed encrypted while live data is retrieved directly from the tracker are not encrypted in transit. With the message format known, the packets captured could be decoded easily. Throughout the entire sniffing process, it is also observed that the Bluetooth MAC address of Fitbit Charge HR remained the same, just as what previous analysis [2] observed. Other than passive sniffing, an attempt was made to directly connect to Fitbit Charge HR without the use of Fitbit Android application. Connection was possible without authentication, but data that can be retrieved from the tracker remains limited, unless the connection was made from a device running the Fitbit Android application in the background to allow for proper authentication.

We did not find new vulnerabilities on the Fitbit Charge HR, although there are indeed ways to anonymously collect real-time data about the user.

Future work could include emulating the authentication process given that the keys used can be extracted, making it possible to independently read live data from a Fitbit device. One major obstacle would be to obtain the necessary keys required to perform the MAC authentication challenge with Fitbit Charge HR as it is not transmitted over the air. The attacker also has to be in close proximity with the bearer of the device since Fitbit Charge HR has a range of 6m for BLE. Another area for analysis would be the firmware of Fitbit Charge HR, which could be captured during a firmware update session.

ACKNOWLEDGEMENT

This project would not have been possible without the guidance of Mr Koh Ming Yang, the mentor for this project.

We would like to take the opportunity to express our heartfelt gratitude towards Mr Tan Jia Jun and Mr Timothy Goh for their assistance and suggestions. Also, we would like to thank Mr Sing Jiun Shin for loaning us the necessary equipment.

REFERENCES

[1] M. Rahman, B. Carbunar, and M. Banik, “Fit and vulnerable: Attacks and defenses for a health monitoring device”, CoRR, vol. abs/1304.5672, 2013.

[2] B. Cyr, W. Horn, D. Miao, M. Specter, “Security Analysis of Wearable Fitness Devices (Fitbit)”, MIT, 2014.