

Security Analysis of Wearable Fitness Devices (Fitbit)

Britt Cyr, Webb Horn, Daniela Miao, Michael Specter

Massachusetts Institute of Technology

Cambridge, Massachusetts, U.S.A.

`cyrbritt@mit.edu`, `webbhorn@mit.edu`, `dmiao@mit.edu`, `specter@mit.edu`

Abstract

This report describes an analysis of the Fitbit Flex ecosystem. Our objectives are to describe (1) the data Fitbit collects from its users, (2) the data Fitbit provides to its users, and (3) methods of recovering data not made available to device owners. Our analysis covers four distinct attack vectors. First, we analyze the security and privacy properties of the Fitbit device itself. Next, we observe the Bluetooth traffic sent between the Fitbit device and a smartphone or personal computer during synchronization. Third, we analyze the security of the Fitbit Android app. Finally, we study the security properties of the network traffic between the Fitbit smartphone or computer application and the Fitbit web service.

We provide evidence that Fitbit unnecessarily obtains information about nearby Flex devices under certain circumstances. We further show that Fitbit does not provide device owners with all of the data collected. In fact, we find evidence of per-minute activity data that is sent to the Fitbit web service but not provided to the owner. We also discovered that MAC addresses on Fitbit devices are never changed, enabling user-correlation attacks. BTLE credentials are also exposed on the network during device pairing over TLS, which might be intercepted by MITM attacks. Finally, we demonstrate that actual user activity data is authenticated and not provided in plaintext on an end-to-end basis from the device to the Fitbit web service.

1 Introduction

1.1 Motivation

Ubiquitous computing and pervasive sensors have become increasingly more available to consumers. Combined with the fact that such computers are commonly used to track health and fitness data, the security and privacy issues related to these devices has become incredibly paramount.

Worse, companies like Fitbit do not provide any control of the data, upload it to their personal cloud, and force the user to pay a subscription fee in order to get further analysis. In the end, the user is given very little indication of what data the device or its associated applications are able to collect. Historically, the Fitbit has had numerous security vulnerabilities, some leading to awkward disclosures of user data [1], security bumbles with communication

between the device and the web server [2], and a myriad of issues relating to the device itself [3]. Such security issues are not new to health tracking services (see [4], for example), and seem quite endemic to these types of devices.

1.2 Goals

The goal of this project is to understand how much Fitbit has progressed in adding security to their devices since the slew of vulnerabilities referenced in the above section. Put another way, before consumers consent to wearing a computer, they should have the ability to know what data it collects and how to see it. A major contribution of this project is therefore to generate an understanding of what data Fitbit collects, how it collects it, and how that data is transferred back and forth from the server. Below is an itemized list of intermediate steps and goals:

- Determine what data Fitbit collects about the user
- See what data Fitbit tells the user about, and compare that data to what is collected
- Recover any data which they collect but don't make available
- Map out other possible security and privacy vulnerabilities associated with the system

2 Related work

Before we began tackling the project, we did a general literature review on the subject in order to gain an understanding of the existing tools and resources available. An online blog named “Hacks by Pete” [5] offered us some initial insight into the format of the Fitbit device syncs. The owner of the blog analyzed sync logs that were saved on his computer after a Fitbit One was paired with it. Even though this work gave us confirmation that we were capturing similar data from the Fitbit Flex, it did not provide any useful information beyond that. This is mainly due to the lack of documentation on the blog, as well as the fact that Fitbit has since updated their software to obfuscate the sync logs on the computer as well.

Another useful resource we found was the open-source initiative called “galileo” for Fitbit [6], created by a German developer that wanted to offer software that allows Fitbit devices to sync to the Linux operating system. Developers for this project have done significant work on the format of the sync data, and the communication protocols between the Fitbit device and the mobile or computer application. In particular, they specified header bytes for each distinct Fitbit device [7] and provided many samples of data captures [8]. While this information helps us understand the format of the captured data from our Fitbit Flex devices, it does not attempt to decrypt the data. One developer seemed to have discovered that the encryption used is AES, but does not provide additional information on how this was inferred. As a result, our team had no background information on exactly which method of encryption (if any) is used, and how the encryption keys are determined or exchanged.

In addition to the above resources, we discovered a variety of independent projects that attempted to analyze the Fitbit communication protocol. Some attempts with older (and

discontinued) Fitbit devices seem to be quite successful [9], while others have been quite futile [10]. All in all, it appears that Fitbit had suffered from significant security attacks in their older models (Fitbit Tracker) due to lack of encryption or any secure communication protocols. Since then, Fitbit has drastically tightened security on both their devices (Fitbit Flex, One and Zip) and applications, making it more difficult for device owners to obtain unauthorized access to the data captured by the devices.

Apart from research on the Fitbit device, we also learned about the Bluetooth protocol. A previous security project, completed as a part of the 6.858 Computer Systems Security in Fall 2012 at MIT, studied the vulnerabilities in the Bluetooth protocol [11]. It served as a very useful introductory document to understanding Bluetooth communication, as well as maximizing usage of the Ubertooth device we have purchased.

3 System overview

Fitbit devices are designed to be worn by their owners all day. To accommodate this use-case, Fitbit devices are designed to buffer activity data locally on the device. Occasionally, the user must synchronize their device with the Fitbit service.

An app for Android, iOS, and desktop platforms is available to perform the synchronization. During synchronization, the Fitbit application forwards the user’s buffered activity data to Fitbit-operated servers over the Internet, where the activity data is warehoused. The activity data is not persisted on the smartphone or personal computer—user data is fetched from the Fitbit cloud service during each synchronization.

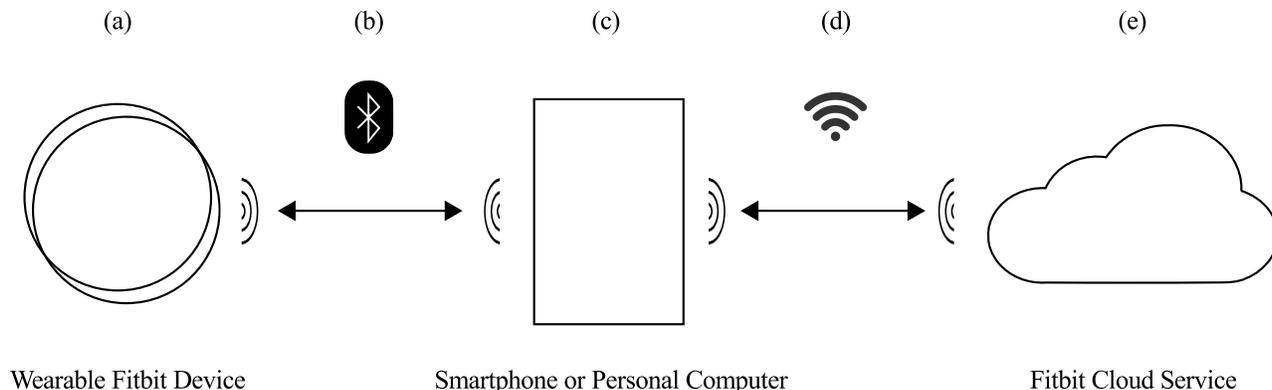


Figure 1: The Fitbit system components. We partition the attack surface into five regions of analysis, (a), (b), (c), (d), and (e).

Synchronization between Fitbit devices and smartphones/personal computers is performed over Bluetooth. In particular, the BTLE [12] (Bluetooth Low Energy) protocol is used. Synchronization between smartphones/personal computers and the Fitbit service occurs in an encrypted session over the Internet.

In figure 1, we partition the Fitbit system components into five regions of analysis. The remainder of this paper is structured as an analysis of these components in turn. Section 4 describes our analysis of the hardware of the Fitbit Flex device (a). Section 5 describes our analysis of the Bluetooth communication (b) between the Fitbit Flex and our Nexus 5 during

pairing and synchronization. Section 6 provides an analysis of the Fitbit Android app (c), while section 7 provides an analysis of the network communication between the Fitbit device and the Fitbit web service (d). We defer an analysis of the Fitbit API and web service (e) to future work in section 8.

4 Device Analysis

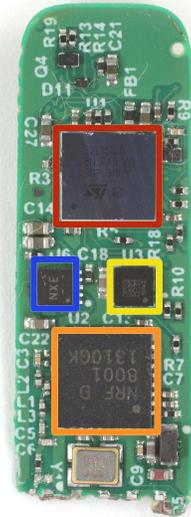


Figure 2: Front of the Fitbit motherboard. Red is the main cpu, orange is the BTLE chip, blue is a charger, and it is unclear what the yellow chip is [13]

According to an excellent tear-down of the Fitbit provided by the repair site ifixit, the main chip on the motherboard is an ARM cortex processor; the STMicroelectronics 32L151C6 [13]. Further research into this specific chip revealed that the device itself has what is commonly called a JTAG interface, which allows researchers and developers to debug the device and possibly dump firmware. Unfortunately, the device also has what is called a JTAG fuse, which is essentially a jumper on the device which, once shorted, will wipe the firmware if a debugger is installed.[14] We confirmed that this was the case by disassembling a broken Fitbit and examining the back of the chip. Because of the difficulty associated with debugging the device, especially since the JTAG interface had been blocked, we decided that focusing on other parts of the device would be more effective.

5 Bluetooth analysis

Since the syncing process from the Fitbit Flex device to the mobile application or computer application is completed via the Bluetooth protocol, we start our experiment by attempting to actively sniff Bluetooth traffic. Commercial tools for this can cost up to thousands of

dollars, though fortunately for us a recent open-source project named “Ubertooth”¹ offers such a tool at a much more affordable price. The Ubertooth hardware plugs into any computer via USB port and the project offers existing software tools that can monitor, track and capture Bluetooth traffic. More importantly, since the Fitbit operates not through the standard Bluetooth 4.0 protocol, but rather BTLE, Ubertooth is able to sniff BTLE traffic as well.

By using the “ubertooth-btle” utility tool that comes as a part of the Ubertooth suite, we were able to capture all traffic to and from the Fitbit Flex device, and follow its communications with the Fitbit Application over multiple Bluetooth hop channels (for background on how Bluetooth communication frequently hops over multiple channel, see[11]). The captured Bluetooth packets are stored to disk in pcap format, which can then be viewed in Wireshark² by installing the corresponding BTLE plug-in.

5.1 Results

Using the above mentioned methods, we were able to capture all Bluetooth traffic that occurs during initial device pairing as well as all subsequent data syncs to the server. From initial analyses of the data, we discovered that the Fitbit Flex responds to broadcasts from any Bluetooth device in range. This enables us to obtain the private address of the Fitbit Flex device, and private addresses of all other BTLE devices nearby, most of them other Fitbit devices. According to the specifications for BLTE, one of the best features it has is privacy-awareness, which allows developers to frequently change the private addresses of devices in order to avoid tracking [15]. It has been suggested that such a feature should be used actively by health monitor devices to preserve privacy of users. However, over the course our experiment we did not observe changes in the private addresses of any of our Fitbit Flex devices. The fact that Fitbit chose to not use this privacy feature of BTLE could lead to potential breaches of privacy as it allows third-parties to track activities of specific users. In addition, since the Fitbit application reports private addresses of nearby devices to the server, and these private addresses never change, it could mean Fitbit is able to construct a profile on each user’s surroundings and activity patterns.

We also explored a known vulnerability addressed in [16], where the BTLE key exchange can be captured via Ubertooth, thereby exposing the encryption key. The author, Mike Ryan, developed a tool named crackle³ that automates this process based on captured Bluetooth traffic in Wireshark pcap format. We tried to exploit this vulnerability by capturing the device pairing process from Fitbit Flex to the Fitbit mobile application. However, the key exchange was not identified by crackle, suggesting that Fitbit did not follow the standard BTLE key exchange protocol in order to obfuscate the encryption key. We did not pursue further brute-force methods of identifying the key as it would be time-consuming and beyond what the project timeline allows.

¹<http://ubertooth.sourceforge.net/>

²<http://www.wireshark.org/>

³<https://lacklustre.net/projects/crackle/>

6 Android app analysis

The next attack vector that we explored was the Fitbit android app. In this section, we describe our experimental setup, including the hardware and software used to decompile, analyze, and modify the app, and the results found after analysis.

6.1 Methods

We used a Nexus 5 Android phone as a testbed for our android app modifications. To decompile, modify, and rebuild the Android app, we used a suite of tools running on OS X.

The first step in reverse engineering the android app is to acquire the application itself. For this, we used apk extractor [17]. Apk extractor emails an installed app to you as an attachment.

The extracted application (`.apk` file) is a zipped and signed archive of application resources and a statically linked program. To examine the program, we unzipped the archive and analyzed the `classes.dex` file contained therein.

`classes.dex` contains the Dalvik [18] machine code of the fitbit application. By itself, machine code is difficult to understand. Therefore, we adopted two common approaches used to reverse engineer dalvik code.

First, we can decompile the Dalvik code to Java using dex2jar [19]. This is a one-way operation—it is simple to generate java source files from a Dalvik program image, but it is difficult to compile the resulting Java source back into a Dalvik image. We therefore use the Java source to get hints about what the application is doing.

Second, we can disassemble the Dalvik code into Smali [20], an assembly language for Dalvik. There is a simple one-to-one mapping between the Dalvik machine code and Smali assembly, so it is easy to modify the Smali source and repackage the application for use on our Nexus 5. We use baksmali [20] to disassemble the application, and Smali to re-assemble our modified code.

To repackage and re-install our modified version of the Fitbit app on our Nexus 5 testbed, we replaced `classes.dex` in the original archive with the result of re-assembling the modified smali code. We then rezip the archive aligned at 4 bytes with `zipalign` and resign the `apk` using `keytool`. Both `zipalign` and `keytool` are part of the android development tools package [21]. We use `adb`, also part of the android development tools package, to reinstall the modified package onto our Nexus 5 testbed.

6.2 Analysis

After decompiling the android app and trying to make sense of the code, we discovered “Live data mode.” “Live data mode” is a feature of Fitbit devices that displays live metrics on the application display while connected to the Fitbit device. We determined that “live data mode” operates on unencrypted data. Initially, we hypothesized that we could modify the unencrypted data in transit (on the phone) and mount a replay attack. This proved to be unsuccessful, however, because activity information transmitted in “live data mode” is never committed to the Fitbit service online, the authoritative record of data—it is just buffered on the device for display purposes.

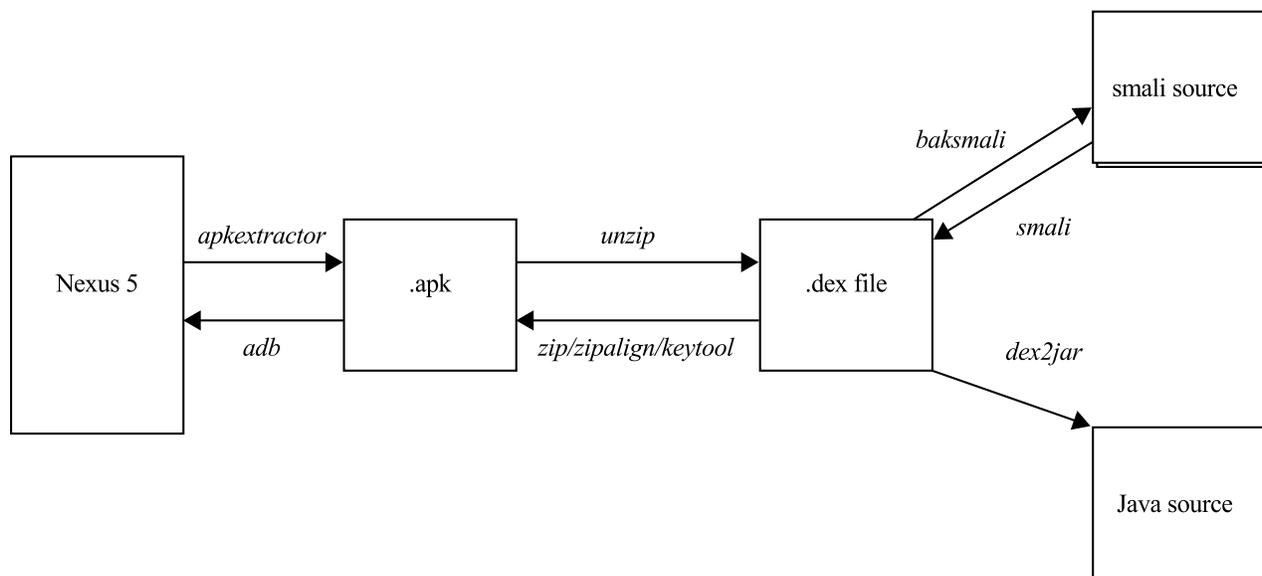


Figure 3: Android toolchain used to modify the Fitbit app.

While tracking down the mechanism of “live data mode,” we noticed a method in the `logging` class that is repeatedly invoked. Upon inspection, however, we noticed that the method did nothing.

We suspect that the developers commented or deleted out the logging call before shipping the app. We inserted logging statements in the function and recompiled the app. The new logged data was revealing. In particular, we saw step data in finer granularity than Fitbit provides. The app only shows steps aggregated to 5 minute intervals while the wifi packets have 15 minute intervals. The logging statement contains step data at 1 minute granularity.

This is data which is not available (or exportable) by directly using the app. Once the logging statements were functioning, we decided to further investigate the logs provided by “Live Data mode” classes. We observed live data packets which are similar to those shown in figure 4.

One important observation we made was that live data packets are nearly identical, indicating that encryption for live data packets is not used (or at least does not employ randomized encryption). We decided to try to exploit this. We modified the part of the code where the app reads the live data, changing the step count in the packets. This was partially successful — we were able to control the value displayed on the mobile device. In one instance we were able to convince the app that the device had only counted 7 steps all day. However, this data did not propagate to the server. We believe that the live data is only there so that the user can get immediate updates and that the data it sends is duplicated in the synchronization packets which are better protected.

From here, we shifted focus onto determining how the Bluetooth protocol worked between the phone and flex. From examining further log statements, we were able to discern much of how the Bluetooth protocol works. The device authenticates with the app by computing a MAC over random bits, using a CBC-MAC with the XTEA block cipher. There is no

```

.method public static c(Ljava/lang/String;Ljava/lang/String;)V

    .registers 2

    .prologue
    .line 108

    return-void
.end method

```

```

.method public static c(Ljava/lang/String;Ljava/lang/String;)V

    .registers 2

    .prologue
    .line 108

    invoke-static {p0, p1}, Landroid/util/Log;d(Ljava/lang/String;
        Ljava/lang/String;)I

    const-string v0, "HACKING"
    const-string v1, "FUNCTION 1"
    invoke-static {v0, v1}, Landroid/util/Log;d(Ljava/lang/String;
        Ljava/lang/String;)I

    return-void
.end method

```

Figure 4: Our modifications to the logging function. The top figure displays the logging function shipped in Fitbit’s production Android app. The bottom figure is our modification to the logging function.

further authentication as far as we could tell.

There are two types of packets that the flex sends to the phone. The first type is a control packet. This is a packet in which the first byte is -64. The rest of the packet maps to an opcode which changes the state of the phone in some way. The second type of packet is a data packet. These are where the interesting information is sent. The packets are all 20 bytes long except for the last one which can vary. The phone verifies these bytes before using them.

We believe that this communication is encrypted. There were no obvious patterns in the packets, so it is likely using some randomized encryption. However, the fact that timestamps in the packets are at a fine granularity could make the entirety of the packet look random, there could be patterns that the timestamps are obfuscating. Because the log statements say that there is a randomized authentication challenge and the number of bytes sent is verified, we concluded it was unlikely that we would simply be able to run a replay attack to get extra steps.

```

05-02 14:57:08.049: D/ServerGateway(13048): RESPONSE:
{"activities-steps":
  [{"dateTime":"2014-05-02","value":"1768"}],
"activities-steps-intraday":
  {"dataset":
    [{"time":"00:00:00","value":0},{"time":"00:01:00","value":0},
    {"time":"00:02:00","value":0},{"time":"00:03:00","value":0},
    {"time":"00:04:00","value":0},{"time":"00:05:00","value":0},
    {"time":"00:06:00","value":0},{"time":"00:07:00","value":0},
    {"time":"00:08:00","value":0},{"time":"00:09:00","value":0},
    {"time":"00:10:00","value":0},{"time":"00:11:00","value":0},
    {"time":"00:12:00","value":0},{"time":"00:13:00","value":0},
    {"time":"00:14:00","value":0},{"time":"00:15:00","value":0},
    ...
  ]
}

```

Figure 5: One minute granularity for step counts during first 15 minutes of the day

```

05-02 14:57:05.689: D/ConnectionState.Airlinking(13048):
New live-data value arrived:
[126, -22, 99, 83, -24, 6, 0, 0, -40, -100, 19, 0, 96, 4, 0, 0]
...
[126, -22, 99, 83, -24, 6, 0, 0, -40, -100, 19, 0, 96, 4, 0, 0]
...
[13, -21, 99, 83, -24, 6, 0, 0, -40, -100, 19, 0, 97, 4, 0, 0]

```

Figure 6: Live data packets which are very similar

7 Network analysis

7.1 Methods

The Android application communicates with the Fitbit web service over an encrypted TLS connection. In order to view the traffic, we used Charles Proxy [22]. Charles proxy enables you to examine traffic originating from your smartphone by running as a proxy on your machine. The proxy intercepts all TLS traffic and replaces the service's TLS certificate with a custom Charles certificate. After adding Charles as a trusted authority on the phone, we are able to successfully view traffic during a session on the proxy.

7.2 Analysis

We started from the initial pairing of the flex with the phone. The phone requested images from the server which it displayed as instructions during the pairing process. However at one step late in the pairing process, the phone requested data from an innocuous message.xml and the server returned with a large Javascript file. Simultaneously, the android log posted warning messages that the app was running insecure content. This is a possible attack vector.

It seems the app is not doing any checking of the Javascript that it is receiving over wifi.

Another interesting packet that we intercepted during the pairing process was the BTLE key.

```
{
  "btleClientAuthCredentials": {
    "authSubKey": "877CB60D0E417DE2905CC214BFB91B77",
    "nonce": 88987889
  }
}
```

Figure 7: btle credentials

After the pairing process, most of the wifi traffic is requesting data to display from the server. The step data is all aggregated to 15 minute intervals the same as the app displays. The phone gets other data from the server which seems excessive, but this did not seem to be a problem. To send the step data that the tracker collects, the phone posts a megadump⁴ to Mixpanel (an analytics company). This server is not the same as the ones from which the phone requests data. The megadump which presumably contains all the fitness data, appears to be base64 encoded. There is definitely some structure to it, however, because the megadump always begins with the same substring of "KAIAAA." We could have attempted some cryptanalysis here, but we focused more on the Bluetooth side of the system because the decompiled source was clearer for that.

8 Summary and Future work

A quick summary of our results thus far reveals the following:

- From the analysis of Bluetooth, we confirmed our findings from Section 5 that the private address of the device does not change. As mentioned, this could allow for tracking a person based on their Fitbit's Bluetooth advertisement.
- During the pairing process, the phone tells the server about all Fitbits within its range unnecessarily.
- From inspecting the phone, we found logs which contain more data than the phone app provides the user.
- We found that BTLE credentials are sent to the phone from the server in plaintext, albeit over TLS.
- There is also Javascript sent to the phone which may yield an attack vector.

Our results indicate that while the Fitbit security setup generally appears to be sound, there is room for further exploration and analysis. We present avenues for future analysis in terms of the five regions identified in figure 1.

⁴A "megadump" is a base64 encoded data blob that we believe contains the raw data collected from the device

8.1 Fitbit Firmware Capture and Reverse Engineering

Although the JTAG fuse has made debugging the device itself difficult, we believe that it is still possible to gain access to the firmware of the device, and then to reverse engineer and modify that. This could be done by instrumenting the Fitbit application to determine where the firmware is stored on disk or in memory prior to an update. Analysis of firmware images has been a successful vector for possible security vulnerabilities in the past⁵, and would have provided much in the way of information regarding the encryption routines and procedures on the device itself.

Although unencrypted firmware images would be ideal, it is possible for the firmware to be updated in such a way that the firmware would be encrypted until installation on the device. Analyzing the end-to-end communication involved in an update, as we have done with the rest of the communication in this paper, would be an excellent place to start.

8.2 Bluetooth

As we have described above, Bluetooth communication between the Fitbit device and the smartphone or computer application is authenticated with a MAC in CBC mode using XTEA. Because the phone must contain the key used to authenticate communication, we could extract the key during more analysis of the phone. With the key and algorithm, it would become possible for us to forge a MAC and launch a replay attack over Bluetooth.

8.3 Android application

The fact that activity data is not stored on the phone or interpreted in transit makes it difficult to exploit the Android app as a means to acquire user data. However, the Android application maintains an interesting set of sqlite3 tables that we suspect might indicate the existence of a development mode that might store activity data locally. It would be worthwhile to spend more time examining the app to see if it is possible to enable an alternative code path that uses these tables.

8.4 Network analysis

We noted in section 7 that Javascript is transmitted to the application during a pairing transaction. We suspect that this Javascript contains updates to the user interface because of the other resources (images, copy text) included alongside the Javascript. Because the Javascript must be executed on the device, we believe that an interesting attack vector could be to inject our own Javascript into the pairing transaction. The effect of this attack would be mitigated, however, by the fact that the pairing process occurs over SSL.

⁵see [23] and [24] for examples of when reverse engineering firmware has turned up surprising and interesting results

8.5 Fitbit server

Finally, we did not perform an audit of the web security of the Fitbit API or the Mixpanel analytics service interposed between the device and Fitbit’s web service. It would be interesting to examine the Fitbit service’s vulnerability to common security vulnerabilities (like CSRF or XSS attacks), or undocumented private API calls.

9 Conclusion

Recall that our three objectives were to describe (1) the data Fitbit collects from its users, (2) the data Fitbit provides to its users, and (3) methods of recovering data not made available to device owners. We believe that we made significant progress toward each objective.

We determined that Fitbit collects extraneous information about users, including the MAC addresses of nearby Fitbits. We also discovered that the android application has access to step data with granularity down to a minute, but the user interface does not present this. We were able to show this data in the log after modifying the source.

Our most successful analysis technique consisted of reverse engineering the Android application, described in section 6.

Overall, the Fitbit provides a reasonable level of privacy for user data, but we would prefer a design that provided valid users an easy-to-access method for acquiring the full set of data recorded by the device.

References

- [1] L. Rao, “Sexual activity tracked by fitbit shows up in google search results,” 2013. [Online]. Available: <http://techcrunch.com/2011/07/03/sexual-activity-tracked-by-fitbit-shows-up-in-google-search-resul>
- [2] “OpenYou - fitbit and security, or lack thereof,” 2010. [Online]. Available: <http://www.openyou.org/2011/04/18/fitbit-and-security-or-lack-thereof/>
- [3] M. Rahman, B. Carbunar, and M. Banik, “Fit and vulnerable: Attacks and defenses for a health monitoring device,” *CoRR*, vol. abs/1304.5672, 2013.
- [4] M. Al Ameen, J. Liu, and K. Kwak, “Security and privacy issues in wireless sensor networks for healthcare applications,” *Journal of medical systems*, vol. 36, no. 1, p. 93101, 2012. [Online]. Available: <http://link.springer.com/article/10.1007/s10916-010-9449-4>
- [5] Pete, “Fitbit sync decode - part 1,” 2013. [Online]. Available: <http://hacksbypete.blogspot.com/2013/01/fitbit-sync-decode-part-1.html>
- [6] “benallard galileo bitbucket,” 2014. [Online]. Available: <https://bitbucket.org/benallard/galileo>
- [7] “benallard galileo wiki megadumpformat bitbucket,” 2014. [Online]. Available: <https://bitbucket.org/benallard/galileo/wiki/Megadumpformat>
- [8] “Data format analysis openyou/libfitbit wiki,” 2012. [Online]. Available: <https://github.com/openyou/libfitbit/wiki/Data-Format-Analysis>
- [9] M. Rahman, B. Carbunar, and M. Banik, “Fit and vulnerable: Attacks and defenses for a health monitoring device,” 2013. [Online]. Available: <https://www.petsymposium.org/2013/papers/rahman-health.pdf>
- [10] T. RA, “Fitbit flex under linux.” [Online]. Available: https://docs.google.com/file/d/0BwJmJQV9_KRcSE0ySGxkbG1PbVE
- [11] E. Chai, B. Deardorff, and C. Wu, “6858 hacking bluetooth,” 2012. [Online]. Available: <http://css.csail.mit.edu/6.858/2012/projects/echai-bendorff-cathywu.pdf>
- [12] “How do fitbit trackers sync with android devices?” [Online]. Available: <https://help.fitbit.com/customer/portal/articles/987748-how-do-fitbit-trackers-sync-with-android-de>
- [13] “Fitbit flex teardown - iFixit.” [Online]. Available: <http://www.ifixit.com/Teardown/Fitbit+Flex+Teardown/16050>
- [14] ST.com, “STM32L15xx6/8/B device manual.” [Online]. Available: <http://www.st.com/st-web-ui/static/active/en/resource/technical/document/datasheet/CD00277537>
- [15] C. Gomez, J. Oller, and J. Paradells, “Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology,” *Sensors*, vol. 12, no. 9, pp. 11 734–11 753, 2012.

- [16] M. Ryan, “Bluetooth: With low energy comes low security,” *Usenix*, 2013. [Online]. Available: <https://www.usenix.org/conference/woot13/workshop-program/presentation/ryan>
- [17] K. Heeseok, “Apk extractor.” [Online]. Available: <https://play.google.com/store/apps/details?id=net.sylark.apkextractor>
- [18] B. DeLacey, “Google calling: Inside android, the gphone sdk.” [Online]. Available: <http://www.onlamp.com/pub/a/onlamp/2007/11/12/google-calling-inside-the-gphone-sdk.html>
- [19] “dex2jar: Tools to work with android .dex and java .class files.” [Online]. Available: <https://code.google.com/p/dex2jar/>
- [20] “smali: An assembler/disassembler for android’s dex format.” [Online]. Available: <https://code.google.com/p/smali/>
- [21] “Android developer tools (adt).” [Online]. Available: <http://developer.android.com/tools/index.html>
- [22] K. von Randow, “Charles: Web debugging proxy application.” [Online]. Available: <http://www.charlesproxy.com/support/>
- [23] J. Zaddach and A. Costin, “Embedded devices security and firmware reverse engineering.”
- [24] “Reverse engineering a d-link backdoor - /dev/ttyS0.” [Online]. Available: <http://www.devttys0.com/2013/10/reverse-engineering-a-d-link-backdoor/>