MOBILE APPLICATIONS: A BACKDOOR INTO INTERNET OF THINGS?

Axelle Apvrille
Fortinet, France

Email aapvrille@fortinet.com

ABSTRACT

While the Internet of Things blossoms with newly connected objects every day, the security and privacy of these objects is often of a lesser priority due to market pressure. To assess their effective security status – and improve it – researchers need to reverse engineer them. Unfortunately, this is not an easy task thanks to the wide variety of smart objects: they often use custom hardware, firmware, operating systems and protocols, meaning that each reverse engineering can be like starting from scratch in a brand new domain – a time-consuming process.

In this paper, we address this issue and propose an easier way to start reverse engineering smart objects. The idea consists of focusing not on the object itself, but on the mobile applications that come with it in numerous cases. We show that this methodology gives good results and illustrate it using three smart objects: a connected toothbrush, a smart watch and a home safety alarm.

1. INTRODUCTION

The Internet of Things (IoT), defined in the *Oxford English Dictionary* as ‘a proposed development of the Internet in which everyday objects have network connectivity, allowing them to send and receive data’, is invading our lives. In 2014, there were approximately 2 billion IoT devices. That is more than the number of laptops and desktop PCs combined (1.5 billion), and comparable with the number of smartphones (1.8 billion) [1]. Some of the most common IoT objects are smart watches (note that, in Q4 2015, the shipment of smart watches overtook the shipment of Swiss watches [2]), fitness wristbands (for humans generally, but also for cats and dogs with devices such as *Otto Pet* systems), smart TVs and smart glasses. But IoT devices are actually found in a wide range of domains:

- Entertainment e.g. *Archos* music beany, *Mattel*’s *Hello Barbie* connected doll.
- High tech e.g. *Recon Instruments*’ augmented reality snow mask, *Narrative*’s wearable cameras, tweeting house.
- Fashion e.g. *Volvorii’*s connected high heel shoes or garments.
- Agriculture e.g. cow insemination.
- Health and safety e.g. *Netatmo June* skin exposure detector, *Vigo*’s drowsing detector, *Glow-Cap* medication reminder caps.
- Etc.

Yet, there are numerous concerns over the state of the privacy and security of these objects. Under market pressure, attractive connected objects are often sold too early without having undergone proper security review, and sometimes even without correct security design. For instance, an *HP* study [3] found that 90% of IoT devices collected at least one piece of personal information, 70% of devices used an unencrypted network, and six devices out of 10 with UI were vulnerable to issues such as XSS and weak credentials. Acknowledging this status, *IDC* predicted that 90% of IT networks would have had an IoT-based security breach by December 2016 [4]. Some of the fears have even reached consumers: *Accenture Consulting* conducted a survey of 28,000 individuals in 28 different countries and reported that 47% of consumers had privacy and security fears over IoT [5].

Myth or reality? To assess the security and privacy of smart objects, researchers need to closely analyse their implementation. As technical documentation is seldom available, the first step usually consists of reverse engineering the device in question. Reverse engineering is never an easy task, but it is particularly difficult for IoT devices because (i) they use lesser known, more specific components, and (ii) because smart objects are very different from each other. The former requires expertise on lesser known domains (e.g. operating systems such as *Riot* [6], *Contiki* [7], *Brillo* [8]). The latter means that reversing one smart object does not provide significant helpful experience for reversing a different one. For example, the reverse engineering of a smart watch has little in common with the reverse engineering of a connected toothbrush. Thus, experience gathered during the first task is of little help to the second.

This is precisely the issue this paper addresses. I propose a simple methodology that I have used in the field on several occasions and which proved to be useful. This methodology makes the first few steps of reversing easier, and consequently helps the researcher to understand the device more quickly. It provides valuable information which can make further reverse engineering more focused and better targeted.

The paper first discusses prior work on this topic (section 2), then explains the methodology (section 3). The following sections illustrate the methodology using the examples of three different smart objects: a connected toothbrush (section 4), a smart watch (section 5), and a home safety alarm (section 6). Finally, we consider the consequences of this methodology.

2. STATE OF THE ART

Although popular, security research on IoT is still in its early days, with far fewer publications than in other fields such as OS security. We can cite work on *NEST* thermostats [9],*WeMo* power sockets [10], vulnerabilities in health infusion pumps [11], injection of arbitrary code in *Fitbit Flex* dongles [12], Bluetooth scanning [13], baby monitors [14], etc. Those pieces of research provide interesting hints for reverse engineering IoT devices, but they also illustrate how different smart objects are from each other, and the amount of work the researchers had to undertake in order to understand them.

[14] compares the task to a CTF (Capture the Flag), which seems quite appropriate because each challenge is different and time

consuming, and there is no immediate solution available on the web.

More academically, a few researchers have proposed methodologies [15] or automated approaches [16, 17] to assist in the reversing of firmware. [15] draws a checklist of the different attack surface areas on IoT. The list is helpful for audit/pen-testing as an aid to not to forgetting certain areas. However, it is not intended to help with the reverse engineering, merely aimed at covering all aspects. As for automated frameworks and tools, they are promising, but still in their early stages, thus difficult to use in practice.

3. METHODOLOGY

To ease the reverse engineering of IoT devices, this paper proposes the following methodology. Instead of directly reversing smart objects themselves, the methodology consists of focusing first on reversing the mobile applications that come with them.

Many connected objects come with related mobile applications to control, supervise or interact with them. For example, *Meian* safety alarms come with an *Android* companion application to help the end-user start, stop, get status or set zones for the alarm. Similarly, *Beam* toothbrushes come with an *iOS* or *Android* application that communicates with the smart toothbrush, etc. When such a mobile application is available, we argue it is a good idea to look into it:

1. **Simpler**. There are many tools for reversing mobile applications (e.g. *apktool*, *baksmali*, *clutch*, *IDA Pro*). Anti-virus analysts use these tools regularly to inspect mobile viruses and thus there is a support community around them.
2. **Security**. IoT vendors are often tempted to develop these mobile applications for marketing reasons (because they are attractive to the end-user) but don't integrate them as thoroughly in their security designs (if there are any). Consequently, the mobile applications often provide access to not-so-obfuscated source code, and in worse cases, security vulnerabilities which compromise the smart object itself, as we will see in section 6.
3. **Fallback**. If the analysis of the mobile application is insufficient, the researcher can always fall back to the reversing of the smart object as a second step. It is quite likely that the information gathered during the first stage will help perform a more focused reverse engineering in the second stage.

4. BEAM TOOTHBRUSH

4.1 Overview

Smart toothbrushes have existed for a while, but received particular media attention in 2015 when *Beam Technologies* mentioned it would be starting a dental insurance plan around its connected devices [18]. Each *Beam* toothbrush is attached to a dental insurance plan, with its own particular offers (e.g. free toothpaste) and affiliated dentists.

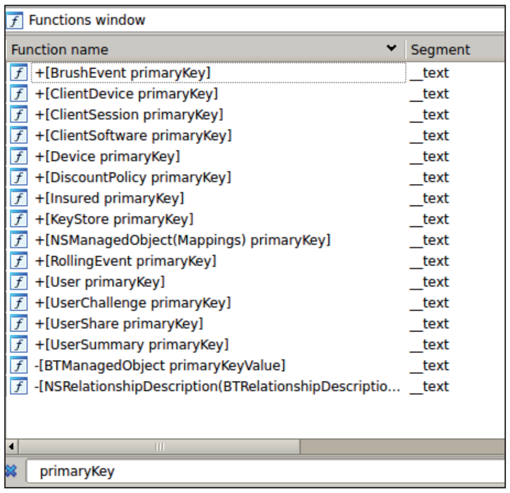
Compared to an insulin pump or pacemaker, a toothbrush probably does not handle the most sensitive health data. Nevertheless, the connection with an insurance policy raises a few questions [19], and it certainly is interesting to investigate how the toothbrush works, whether from an education point of view or for security and privacy concerns.

The toothbrush does not come with *any* technical information apart its commercial specifications [20]: Bluetooth LE 4.0 (BLE), sonic motor, size and colour. There is no user forum, no developer community, and no academic publication. Unfortunately, this is common for IoT devices: we have to start from scratch with no information.

One option would have been to perform a hardware tear down of the toothbrush: open it, get to the electronic components, probe for test points, etc. Instead, we decided to focus on the *iOS* and *Android* mobile applications that come with the device.

4.2 Reversing the iOS application

The reversing of the *iOS* application is helpful in terms of architecture. The application uses a sqlite database named BeamBrushData.sqlite, which contains several tables such as BrushEvent, ClientDevice and ClientSession (see Figure 1). The tables can be listed by searching for the keyword 'primaryKey' in function names.



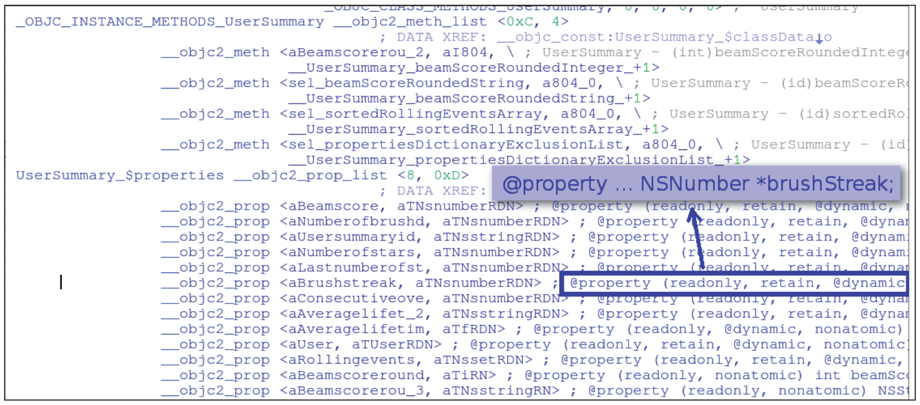
Function name	Segment
+[BrushEvent primaryKey]	__text
+[ClientDevice primaryKey]	__text
+[ClientSession primaryKey]	__text
+[ClientSoftware primaryKey]	__text
+[Device primaryKey]	__text
+[DiscountPolicy primaryKey]	__text
+[Insured primaryKey]	__text
+[KeyStore primaryKey]	__text
+[NSManagedObject(Mappings) primaryKey]	__text
+[RollingEvent primaryKey]	__text
+[User primaryKey]	__text
+[UserChallenge primaryKey]	__text
+[UserShare primaryKey]	__text
+[UserSummary primaryKey]	__text
-[BTManagedObject primaryKeyValue]	__text
-[NSRelationshipDescription(BTRelationshipDescriptio...	__text

primaryKey

Figure 1: SQL tables used by the mobile Beam Brush application.

The contents of each table are described by functions named mappings, for example [Insured mappings] for the Insured table.

Knowing the fields of each table is useful for understanding what data is stored and may potentially leak to an adversary. For example, the BTStarCardInfo table contains the variation of stars for an end-user: name, beforeValue, afterValue, starCount, lastTotalStars, totalStars. Stars are virtual points granted to an end-user when (s)he completes given challenges such as brushing his/her teeth for more than two minutes in a row. An attacker can certainly try to modify the values here to gain (undeserved) stars – although it is likely there are other checks, on the remote servers for instance.



```
_OBJC_INSTANCE_METHODS_UserSummary __objc2_meth_list <0xC, 4>
                                   ; DATA XREF: __objc_const:UserSummary_$classData↓o
                __objc2_meth <aBeamscorerou_2, aI804, \ ; UserSummary - (int)beamScoreRoundedIntege
                              __UserSummary_beamScoreRoundedInteger_+1>
                __objc2_meth <sel_beamScoreRoundedString, a804_0, \ ; UserSummary - (id)beamScoreRo
                              __UserSummary_beamScoreRoundedString_+1>
                __objc2_meth <sel_sortedRollingEventsArray, a804_0, \ ; UserSummary - (id)sortedRo
                              __UserSummary_sortedRollingEventsArray_+1>
                __objc2_meth <sel_propertiesDictionaryExclusionList, a804_0, \ ; UserSummary - (id
                              __UserSummary_propertiesDictionaryExclusionList_+1>
UserSummary_$properties __objc2_prop_list <8, 0xD>
                                   ; DATA XREF:
                __objc2_prop <aBeamscore, aTNsnumberRDN> ; @property (readonly, retain, @dynamic,
                __objc2_prop <aNumberofbrushd, aTNsnumberRDN> ; @property (readonly, retain, @dyna
                __objc2_prop <aUsersummaryid, aTNsstringRDN> ; @property (readonly, retain, @dynam
                __objc2_prop <aNumberofstars, aTNsnumberRDN> ; @property (readonly, retain, @dynam
                __objc2_prop <aLastnumberofst, aTNsnumberRDN> ; @property (readonly, retain, @dyna
                __objc2_prop <aBrushstreak, aTNsnumberRDN> ; @property (readonly, retain, @dynamic
                __objc2_prop <aConsecutiveove, aTNsnumberRDN> ; @property (readonly, retain, @dyna
                __objc2_prop <aAveragelifet_2, aTNsstringRDN> ; @property (readonly, retain, @dyna
                __objc2_prop <aAveragelifetim, aTfRDN> ; @property (readonly, @dynamic, nonatomic)
                __objc2_prop <aUser, aTUserRDN> ; @property (readonly, retain, @dynamic, nonatomic)
                __objc2_prop <aRollingevents, aTNssetRDN> ; @property (readonly, retain, @dynamic,
                __objc2_prop <aBeamscoreround, aTiRN> ; @property (readonly, nonatomic) int beamSc
                __objc2_prop <aBeamscorerou_3, aTNsstringRN> ; @property (readonly, nonatomic) NSS
```

Figure 2: IDA Pro showing methods and properties of the UserSummary class. The comments for properties show the exact type of fields.

The information stored in these tables may be valuable to advertisement kits – or worse adware. For example, the Insured table holds the title, first name, middle name, last name, gender and date of birth of the insured user, who is probably an adult of the family (typically father or mother). Then, the User table provides the same information for other members of the family, for example children. Consequently, by analysing the data contained in those tables, a spy can learn the composition of a given family, and display targeted advertisement or sell the email to spam lists.

Besides database structure, the disassembly of the *iOS* application also reveals the structure of classes in the implementation. The objc segment of the mobile application's binary provides a full overview of methods and fields for each class. For instance, Figure 2 shows the fields – called properties – (e.g. beamScore, numberOfBrushDaysLeft) and methods (e.g. beamScoreRoundedInteger) for the UserSummary class. Comments in *IDA Pro* are particularly useful and even provide the signature for methods and type for each field.

From this we learn:

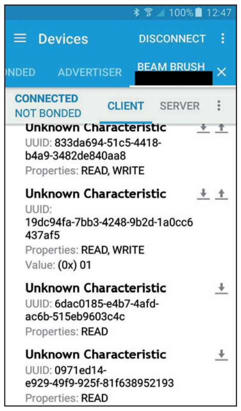
- The toothbrush contains an accelerometer and a gyroscope. Both provide a three-axis vector in BTBrushData class. That is how the toothbrush works out that an end-user is brushing his/her teeth and which quadrant (the mouth is divided into four areas, or quadrants: upper left, upper right, lower right, lower left).
- Firmware Over-The-Air service. The contents of the BTFirmwareUpdater class show there is an over-the-air service for firmware updating. Updating the firmware consists of sending bytes of the new firmware to the toothbrush, until all bytes have been written.

  ```
  char *firmware;
  unsigned int totalLength;
  unsigned int written;
  unsigned int toWrite;
  unsigned int loopCount;
  int state;
  CBService *otaService;
  CBCharacteristic *otaControlPoint, *otaDataPoint;
  ```

- Stars are software only. The number of stars for a given end-user is not stored on the toothbrush itself, but on the mobile phone (and presumably on the remote server databases). Indeed, the classes BTBrushData, BTBrushEvent, Device and ClientDevice do not have a field for stars. From the content of those classes, we learn that the toothbrush is made of a firmware, hardware, serial number, flash, battery level, a motor (whose speed is controllable), a three-axis gyroscope, a three-axis accelerometer, an auto-off timer and a BLE capable chip.

4.3 Reversing the Android application

Reversing the *Android* application can reveal some other details. For example, we know the toothbrush exports several BLE services and characteristics (see Figure 3) but most of these, except the standard ones (e.g. Generic Access), are unknown.



Unknown Characteristic
UUID: 833da694-51c5-4418-b4a9-3482de840aa8
Properties: READ, WRITE

Unknown Characteristic
UUID: 19dc94fa-7bb3-4248-9b2d-1a0cc6437af5
Properties: READ, WRITE
Value: (0x) 01

Unknown Characteristic
UUID: 6dac0185-e4b7-4afd-ac6b-515eb9603c4c
Properties: READ

Unknown Characteristic
UUID: 0971ed14-e929-49f9-925f-81f638952193
Properties: READ

Figure 3: BLE characteristics of the toothbrush.

By reversing the *Android* application, it is relatively easy to find the meaning of each of these characteristics. For example, the code in Figure 4 shows the UUID for the toothbrush's motor speed (which translates into brush strokes per minute) and quadrant buzz (the toothbrush is capable of vibrating when the end-user has spent enough time brushing a given dental quadrant). Known BLE services and characteristics are listed in Tables 1 and 2, respectively.

UUID	Description
00001800-0000-1000-8000-00805f9b34fb	Generic access (standard)
c05fc343-c076-4a97-95d3-f6d3e92a2799	Firmware OTA service
04234f8e-75b0-4525-9a32-193d9c899d30	Beam service
89bae1fa-2b59-4b06-919a-8a775081771d	Probably accelerometer/ gyroscope chip service

Table 1: BLE services of the Beam brush.

UUID	Description
a8902afd-4937-4346-a4f1-b7e71616a383	Boolean indicator for active brushing
267b09fd-fb8e-4bb9-85ccade55975431b	Motor state
3530b2ca-94f8-4a1d-96beaa76d808c131	Current time
833da694-51c5-4418-b4a9-3482de840aa8	Motor speed
19dc94fa-7bb3-4248-9b2d-1a0cc6437af5	Auto-off and quadrant buzz indicators (2 bits)
6dac0185-e4b7-4afd-ac6b-515eb9603c4c	Battery level (2 bytes)
0971ed14-e929-49f9-925f-81f638952193	Brush colour (1 byte)
0227f1b0-ff5f-40e3-a246-b8140205bc49	Accelerometer data (6 bytes)
ed1aa0cf-c85f-4262-b501-b9ddf586a1db	Gyroscope (6 bytes)
cf0848aa-ccdb-41bf-b1e1-337651f65461	Button state

Table 2: Most interesting BLE characteristics of the Beam brush.

With this information, we are able to control the auto-off and quadrant buzz features using our own implementation.

Controlling the features consists of:

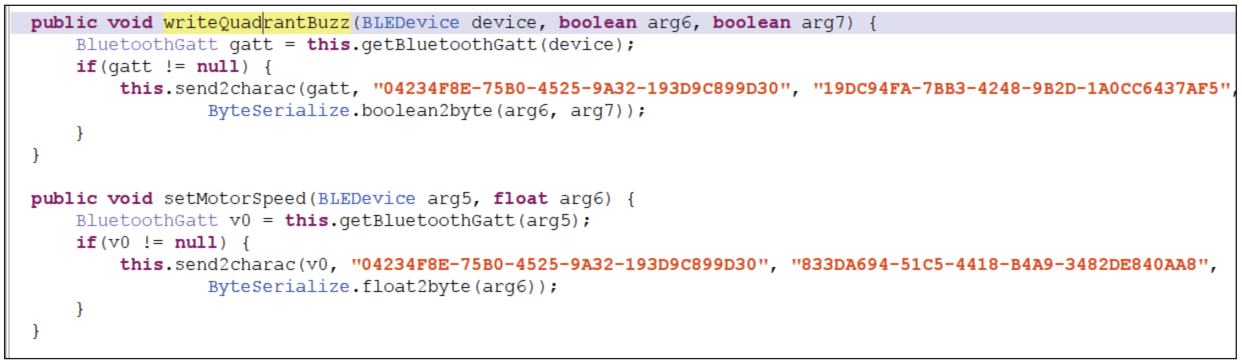
1. Initiating a BLE connection with the toothbrush.
2. Writing the byte value to the appropriate characteristic UUID. The least significant bit controls the quadrant buzz, the second bit controls the auto-off.
3. Disconnecting from the toothbrush.

The BLE commands may be sent using libraries such as gattlib [21] above bluez (Bluetooth stack implementation) and a simple BLE USB dongle.

4.4 Pros and cons

Hacking discoveries for the *Beam* brush are summarized below, with comments on the efficiency a strategy which starts with the reversing of mobile applications.

- **Presence of a gyroscope and accelerometer**. A hardware tear down of the toothbrush would have achieved the same result with more or less difficulty depending on how well the chips are sealed or packaged. Obviously the advantage of reversing a mobile application is that we do not need to open the toothbrush, and do not risk potentially ruining it. The disadvantage is that we are not able to tell the brand and model of the components, and thus their electronic specifications.
- **Existence of a firmware updating service**. Such result would have been difficult to obtain via other means (perhaps by listening to the BLE traffic).
- **Number of stars not stored on the brush itself**. This would have been difficult to find by hardware investigation, probing or BLE scanning.
- **Implementation design**. Obviously, there is no way to get this without disassembling the code. The only part we do not see however is the *hardware* design.
- **Identification of BLE services and characteristics**. This would be feasible using a BLE scanner application (e.g. nRF Master Control Panel [22]), but would take much longer because identification must be guessed by trying various values and noticing the difference in behaviour on the toothbrush.



```
public void writeQuadrantBuzz(BLEDevice device, boolean arg6, boolean arg7) {
    BluetoothGatt gatt = this.getBluetoothGatt(device);
    if(gatt != null) {
        this.send2charac(gatt, "04234F8E-75B0-4525-9A32-193D9C899D30", "19DC94FA-7BB3-4248-9B2D-1A0CC6437AF5",
                ByteSerialize.boolean2byte(arg6, arg7));
    }
}

public void setMotorSpeed(BLEDevice arg5, float arg6) {
    BluetoothGatt v0 = this.getBluetoothGatt(arg5);
    if(v0 != null) {
        this.send2charac(v0, "04234F8E-75B0-4525-9A32-193D9C899D30", "833DA694-51C5-4418-B4A9-3482DE840AA8",
                ByteSerialize.float2byte(arg6));
    }
}
```

Figure 4: BLE characteristics for motor speed and quadrant buzz.

5. SMARTWATCH

5.1 Architecture

We experimented with *Sony*'s *Smart Watch 2*, also known as '*SW2*'.

Unlike the *Beam* brush, there is a lot of technical and developer information available for this smart watch, because *Sony* actually encourages developers to write new applications for it. *Sony* consequently provides an API, documentation, examples and tutorials. The smart watch features an STM32F439 SoC (which includes an *ARM* Cortex-M4 and crypto accelerators), a light sensor, an accelerator, support for NFC and Bluetooth 3.0 (note this is different and not compatible with Bluetooth Low Energy), and a LiPo battery. It runs *Micrium*'s µC/OS-II real-time operating system.

Knowing this, an expert of µC/OS-II or *ST Microelectronics* SoC could certainly have continued the investigation on those parts of the device. In this paper, we assume the researcher does not have access to such experts, and instead we focus on the use of the smart watch. To use the smart watch, at least two *Android* applications must be installed: an application named *Smart Connect* and another one called *SmartWatch 2 SW2*. It is precisely those applications we propose to inspect.

To understand them, it is important to understand *Sony*'s terminology. For *Sony*, a smart watch is, more generically, known as a *smart accessory* because there are other types of accessories, such as headsets. A smart watch 'application' (we will see later there is actually no such thing) is known as a *smart extension*.

To create a new smart extension, a developer compiles his/her code with *Sony*'s Smart Extension API. This creates a real *Android* application (an .apk that researchers can reverse with standard tools such as *apktool*, *baksmali* etc.) – but the application will only work if the two applications we mentioned earlier are both installed.

So, to install the smart extension, an end-user actually installs the developer's apk, i.e. an *Android* application. This application is automatically seen by *Smart Connect*, one of the two mandatory applications, and added to the appropriate smart accessory. The new smart extension icon appears on the smart watch.

Note there is no direct installation on the smart watch itself. As a matter of fact, there is *no concept of a smart watch application at all*. Indeed, all the work of the smart extension is performed on the smart phone. The smart watch basically acts as a remote display. The various events and messages the smart extension generates go through the second of the two mandatory applications, *SmartWatch 2 SW2*. This application is actually what *Sony* calls a *host application*, i.e. an *Android* application dedicated to communication with a given smart accessory (in our case, a *SW2*). The various terms are outlined in Table 3 and Figure 5, as some of them are unfortunately misleading.

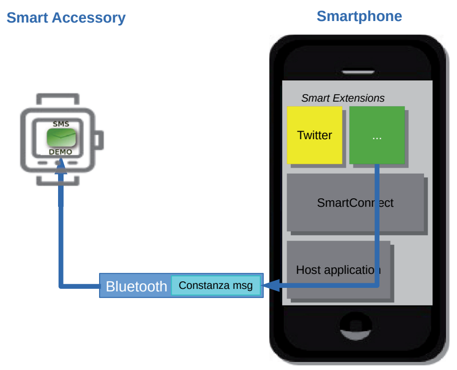


Figure 5: Sony Smart Watch 2 architecture.

5.2 Consequences

With this architecture in mind, the immediate consequence is that any extension on the smart watch can actually be analysed by reversing the code of its *Android* application.

Let's suppose, for instance, we inspect an extension which sends SMS messages. There is nothing to reverse on the smart watch itself. Everything can be done by disassembling the *Android* application where we will see something like smsManager.sendTextMessage(mPhoneNumber, null, message, ...). If smart watch malware were to exist and propagate, anti-virus vendors would merely have to write signatures for the corresponding *Android* applications, which is something they already know how to do.

Name	Description
Host application	Generic term for *Android* applications dedicated to communication with a given smart accessory.
Smart accessory	Generic term for smart watches, smart headsets etc.
Smart Connect	This is one of the two mandatory *Android* applications that must be installed on the smart phone to be able to use the smart watch. It is an official *Sony* application. It manages which extension uses which accessory. Its package name is com.sonyericsson.extras.liveware.
Smart extension	This is an *Android* application, which runs on the phone, but is accessible/controllable remotely from the smart watch. *Sony* provides several extensions (*Twitter* feed, *Facebook* feed, *Chrono*) and encourages developers to create their own.
Smart Watch 2 SW 2	This is the other of the two mandatory *Android* applications for the SW2. Actually, it is the host application for the *SW2* accessory. This is an official *Sony* application. It is curiously named com.sonymobile. smartconnect.smartwatch2.

Table 3: Sony's terminology for the Smart Watch.

The same can be done with the official *SmartWatch 2 SW2* host application. The reversing shows that communication with the remote smart watch is handled by 'Costanza' messages. Those messages consist of a type (type of message), a message identifier and the packed buffer bytes. The packing or unpacking is handled by a native library named 'protocol' (libprotocol.so). Once packed, the messages can be sent by Bluetooth – this is performed using the *Android* API, opening a Bluetooth socket. There are several different types of Costanza messages, such as:

- Battery level indication (*id* = 18). Level is provided as a percentage.
- Factory reset request and response (*id* = 20 or 21).
- 'Force crash' (*id* = 666). The source code shows there is apparently a hidden debug screen where a button 'Force crash on watch' appears. This creates the following message:

  ```
  public RequestForceCrash(int
  newMessageId) {
  super(newMessageId);
  this.type = 666;
  this.mMagic = 0xC057A72A;
  }
  ```

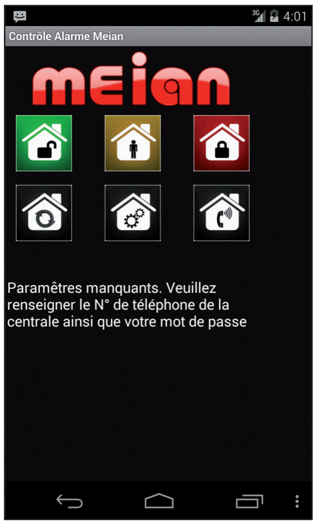
  Note the type 666 and magic value which more or less matches 'costanza' in leet speak.
- Fota request (*id* = 6). This probably means Firmware update Over The Air.
- Sensor data (request or response) (*id* = 127 or 128).
- Swipe indication (*id* = 116).
- Version request and response (*id* = 4 and 5).
- Vibration request (*id* = 129), where the duration the vibration is on, then off and the number of iterations.

The use and identifier of those messages would have been difficult to find by other means (especially the Force Crash message which does not appear in normal circumstances). Again, this proves how profitable the inspection of mobile applications is for IoT.

6. SAFETY ALARM

Meian is a manufacturer of home safety alarms. Some of the company's alarms are remotely controllable via SMS: you can start/stop the alarm remotely, get its current status, enable/disable some zones, etc. The commands must comply to a strict format, and of course, must contain a correct password. To control his/her alarm, the end-user is expected to write an SMS (which complies to the requested format) and send it to the alarm. The alarm receives the SMS, processes incoming messages and replies if okay or not.

As the format for SMS message is strict, *Meian* has implemented an *Android* application that automatically formats the SMS. During set-up, the end-user provides the configuration of his/her alarm: the alarm's phone number, management password, acceptable delay to enter the password, emergency phone number (which is called if an intrusion is detected). Then, the application simply offers buttons to start/stop/get status (etc.) of the alarm (see Figure 6), which is quite handy.



Paramêtres manquants. Veuillez renseigner le N° de téléphone de la centrale ainsi que votre mot de passe

Figure 6: Main screen of the Android application for remote control of one's home safety alarm.

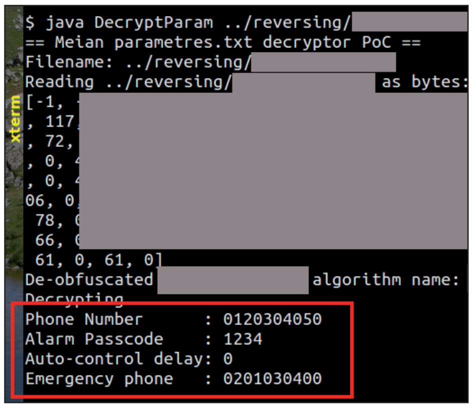
Applying the same methodology, we analyse the mobile application. There are two security issues:

1. **SMS not deleted**. The outgoing SMS remains on the smartphone, unless manually erased. Consequently, if an attacker reads it, s/he gets the alarm's password. Note this issue is present whether the end-user uses the application or not (manually writes the SMS).
2. **Weak protection of alarm's configuration**. The application implements hand-made and unfortunately weak crypto to protect configuration data (phone number, password, emergency phone number and delay). Note that configuration data is sensitive because anybody can control the alarm with it. The cryptographic algorithm can easily be reversed to decrypt any settings (see proof of concept in Figure 7). The vulnerability was reported to *Meian* in 2015 [23], who did not respond. The application remains

Situation: can an attacker retrieve the alarm's password or phone number...	... from a command SMS in the outbox?	... once the SMS messages are erased?
Manual SMS	Yes	No
With *Meian*'s *Android* application	Yes	Yes

Table 4: Security status with or without the home alarm's companion application: unfortunately better without the mobile application!

unpatched on the *Play Store*, and has been downloaded between 1,000 and 5,000 times.



```
$ java DecryptParam ../reversing/[redacted]
== Meian parametres.txt decryptor PoC ==
Filename: ../reversing/[redacted]
Reading ../reversing/[redacted] as bytes:
[-1, [redacted]
, 117
, 72,
, 0,
, 0,
06, 0
 78,
 66,
 61, 0, 61, 0]
De-obfuscated [redacted] algorithm name:
Decrypting
Phone Number       : 0120304050
Alarm Passcode     : 1234
Auto-control delay: 0
Emergency phone    : 0201030400
```

Figure 7: Proof of concept decrypting all major confidential settings of the safety alarm.

The security status of this home safety alarm is summarized in Table 4. Unfortunately, it illustrates one of the worse cases of security for IoT where the use of a companion mobile application actually worsens the security of the device.

7. CONCLUSION

The security analysis of IoT devices can be challenging, mostly because of their variety, and sometimes researchers don't know how or where to start. This paper shows that, in such cases, it is interesting to grab the mobile applications which are meant to communicate with those connected objects, and use them as a starting point. Those mobile applications are quite common for IoT devices.

In this paper, I present three different devices I analysed through their mobile applications: a connected toothbrush, a smart watch and a home safety alarm. In all cases, the strategy quickly revealed implementation design, protocol details and vulnerabilities. Some of those findings could probably have been discovered by other means, e.g. Bluetooth scanning and fuzzing, but it would have taken much longer because we would have had to guess several aspects, whereas mobile application reverse engineering provides certitude.

The fact that IoT and mobile applications are intrinsically tied together has several consequences. First, of course, vendors should spend more time on a secure design and implementation of their devices, but also of the related mobile applications. Note that a quick fix consisting of obfuscating the code will not work: security by obscurity has been discredited on numerous occasions. It is the design and the implementation which needs to be improved. Second, the anti-virus industry needs to be prepared for IoT malware, whether coming from the devices themselves or from mobile applications. This is probably the next malicious battle we will have to fight.

ACKNOWLEDGEMENTS

I would like to thank *Beam Technologies* for providing a free user account for testing purposes. This test account is not attached to any insurance plan. Thanks also to students Axel Ehrenstrom and Soufiane Joumar, and their supervisors, Aurelien Francillon, Ludovic Apvrille and Ruchna Nigam for their initial work on *Sony*'s smart watch.

REFERENCES

[1] BI Intelligence. The internet of everything: 2015. http://uk. businessinsider.com/internetof-everything-2015-bi-2014-12?r=US&IR=T#like-this-decksubscribe-to-bi-intelligencebelow-6.

[2] StrategyAnalytics. Global Smartwatch Shipments Overtake Swiss Watch Shipments in Q4 2015. https://www.strategyanalytics.com/strategy-analytics/news/strategy-analyticspress-releases/strategyanalytics-press-release/2016/02/18/global-smartwatchshipments-overtake-swisswatch-shipments-in-q4-2015#.Vs8cw0J59hH, February 2016.

[3] Hewlett Packard. Internet of things research study. http://h20195.www2.hp.com/V2/GetDocument.aspx?docname=4AA5-4759ENW&cc=us&lc=en, 2015.

[4] IDC. IDC Reveals Worldwide Internet of Things Predictions for 2015, December 2014. https://www.idc.com/getdoc.jsp?containerId=prUS25291514.

[5] Accenture Consulting. Igniting growth in consumer technology. https://www.accenture.com/_acnmedia/PDF-3/Accenture-Igniting-Growthin-Consumer-Technology.pdf.

[6] Baccelli, E.; Hahm, O.; Wählisch, M.; Guìnes, M.; Schmidt, T. RIOT: One OS to Rule Them All in the IoT. Research Report RR-8176, INRIA, December 2012.

[7] Contiki: The Open Source OS for the Internet of Things. http://www.contiki-os.org/.

[8] Brillo. https://developers.google.com/brillo/.

[9] Hernandez, G.; Arias, O.; Buentello, D.; Jin, Y. Smart Nest Thermostat: A Smart Spy in Your Home, 2014. BlackHat US.

[10] Buentello, D. Weaponizing your coffee pot. DerbyCon, 2013.

[11] Hospira Lifecare PCA Infusion Pump 412 Telnet Service weak authentication. CVE-2015-3459. http://www.scip.ch/en/?vuldb.75158.

[12] Apvrille, A. Geek usages for your Fitbit Flex tracker. Hack.lu, October 2015. http://2015.hack.lu/archive/2015/fitbit-hackluslides.pdf.

[13] Wueest, C. Quantified Self – A Path to Self-Enlightenment or Just a Security Nightmare?, 2014. BlackHat Europe.

[14] Wineberg, W. Internet of Things: Hacking 14 Devices, August 2015. DEFCON 23.

[15] Miessler, D. IoT Attack Surfaces, August 2015. DEFCON 23.

[16] Costin, A.; Zaddach, J.; Francillon, A.; Balzarotti, D. A large scale analysis of the security of embedded firmwares, 2014. BlackHat Europe.

[17] Zaddach, J.; Bruno, L.; Francillon, A.; Balzarotti, D. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. Network and Distributed System Security (NDSS) Symposium, NDSS 14, February 2014.

[18] Higginbotham, S. Meet a startup building an insurance business around a connected toothbrush. Fortune. http://fortune.com/2015/06/26/connected-toothbrush-insurance/.

[19] Apvrille, A. Insurance Fraud via Internet of Things, July 2015. http://blog.fortinet.com/post/insurance-fraud-via-internetof-things.

[20] Beam Technologies. https://www.beam.dental/tech.

[21] gattlib. https://bitbucket.org/OscarAcena/pygattlib.

[22] Nordic nRF Master Control Panel. https://www.nordicsemi.com/eng/Products/Nordic-mobile-Apps/nRF-Master-Control-Panel-application/(language)/eng-GB.

[23] Fortinet Discovers Meian Safety Alarm Android Application Weak Credential Encryption Vulnerability. http://www.fortiguard.com/advisory/fortinet-discovers-meian-safety-alarm-android-application-weak-credential-encryption-vulnerability.