SECURITY ANALYSIS OF GARMIN WATCH SOFTWARE

Miłosz Głowaczewski, 202202860

PROJECT REPORT

January 2024

Advisor: Diego F. Aranha

AARHUS UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

SECURITY ANALYSIS OF GARMIN WATCH SOFTWARE

MIŁOSZ GŁOWACZEWSKI

Project Report

Department of Computer Science
Faculty of Natural Sciences
Aarhus University

January 2024

CONTENTS

1 INTRODUCTION

1.1 GARMIN FITNESS WATCHES

In the past years fitness trackers experienced a high growth in popularity. As the technology develops, the devices get more affordable and are able to track more metrics. While it improves the insights into health and fitness tracking, it also means that it collects more sensitive data about the users.

Garmin fitness watches can accompany the users day and night, being able to last even over a week on a single charge. The company has a wide range of options, having released over 100 different watches with third-party apps support [2]. The devices are equipped with a large number of sensors such as heart rate, pulse oximeter, temperature, GPS, compass, accelerometer, gyroscope, altimeter, barometer, light sensor. This allows collecting and calculating an enormous amount of different sensitive metrics. As a result, the potential consequences of unauthorized access to the user's data are severe.

The watches can work independently, collecting and processing the data offline. However, Garmin offers an extensive set of software to improve the user's experience. There are multiple apps available for iPhone and Android phones to manage the watch and the data. This means that the data is being sent over a Bluetooth protocol from the watch to the phone. Additionally, Garmin encourages users to synchronize the data with their cloud.

In case the provided software does not fulfill users' requirements, it is also possible to install third-party apps on the watch. Garmin maintains an ecosystem of tools for the developers to create apps and publish them in a Garmin store. It makes the watch extensible and more universal, but it also allows for a third-party developer to gain access to some of the user data.

The described software offers a powerful toolset for the users to analyse their fitness data. However, it also creates a large number of possible vectors of attack. It is also important to notice that the Garmin software is not open source, which makes it more difficult to analyse its security. Garmin provides a contact form on their website to report security vulnerabilities. Unfortunately, they do not offer a bounty program.

In this report, a broad analysis of different attack vectors has been performed.

Overall, during the analysis, the following privacy and security concerns were found:

- Garmin uses deprecated SHA-1 for app signing
- Android apps installed on the phone, without any permissions can detect that the user uses Garmin devices and the type of the device
- Communication between the third-party apps and their companion apps are in plain text, available to all apps installed on the Android phone. I created a proof of concept app that can eavesdrop on the messages sent to the phone.
- Permission system of the third-party watch apps is not as well separated into different categories as it is in Android, iOS phones.
- Fuzzy testing suggests that it may be possible to escape the virtual machine running the third-party apps.

1.2 SOFTWARE DESCRIPTION

Garmin provides multiple mobile apps to manage the watch and the data.

- Garmin Connect — the main app for managing the watch and the data.
- Garmin Explore — app for managing the watch and the data, focused on the offline usage.
- Garmin Connect IQ Store — app store for third-party apps, that can be installed on the watch.

If the user decides to synchronize the data with the cloud, it is sent to the Garmin Connect services. Then it can be accessed through the website or the mobile app.

Additionally, Garmin has developed a proprietary programming language, Monkey C running in a virtual machine for creating third-party apps. The developers are provided with an SDK[1], containing all the necessary tools for creating and testing the apps. Monkey C language has an integration with Visual Studio Code through the extension[2]. The apps can be tested either on a real device or on a simulator, provided with the SDK. There is an official documentation describing the development process.

After a preliminary analysis, the following scripts and tools were found in the SDK:

- monkeybrains.jar - java archive, compiles and builds the project
- simulator - binary ELF file, simulates a chosen Garmin device

1 https://developer.garmin.com/connect-iq/overview/
2 https://marketplace.visualstudio.com/items?itemName=garmin.monkey-c

- monkeydo - executes Connect IQ executable on the simulator
- Other scripts: barrelbuild, barreltest, connectiq, era, mdd, monkeyc, monkeygraph, monkeydoc, shell, monkeymotion

1.3 testing environment

For the analysis the decision was made to use Linux operating systems as it is a popular choice in the system security community and a large number of materials use it. The chosen distribution was Kali Linux because it comes with already preinstalled toolset.

The testing is executed on the watch Garmin Fenix 7S Pro, released in May 2023. The third-party applications are investigated when using Connect IQ 6.3 SDK, released in August 2023.

Garmin supports both Android and iOS phones for managing the watch. I focused on the Android version of the software. Android is more open and easier to analyse, and I did not have access to Apple devices.

For the purpose of the analysis, when Garmin watches are mentioned, this report focuses on Garmin Fenix watches series 7. Other devices may offer different hardware and software configurations.

1.4 report structure

TODO

1.5 related work

Different attempts at the security analysis of Garmin software are available online. There were attempts to ethically hack a Garmin watch [7,10], analyzing different vectors of attack such as the Garmin Connect website, watch communication, and third-party apps. However, they did not go deep into any of those topics.

Others tried to reverse engineer Garmin's Virtual Machine and do a security analysis of the environment [6, 8]. They were able to get access to the firmware of the watch. With the help of unofficial documentation of the firmware format [9], they proceeded to reverse engineer the bytecode. During the analysis, they determined which parts of the code are responsible for the third-party apps' execution. Thereafter, they found multiple vulnerabilities concerning the app runtime that have been disclosed to Garmin. Some of them affected over a hundred devices for around 7 years.

While they explored the specific topic deeper, there is still a long list of things that could be analyzed in more detail. Previous attempts used an available beta firmware for the analysis and used watches released over 4 years ago. Since then, many new models have been introduced as well as new firmware versions with new features and expanded developer SDK.

Considering the size of Garmin's ecosystem and the number of different components, the existing, publicly available analysis is far from being comprehensive.

2 THREAT ANALYSIS

2.1 FOCUS OF THE ANALYSIS

The amount of software, watch models, and the fact that firmware is being constantly developed makes it infeasible to cover everything in a single publication. I decided to focus on the analysis of the third-party apps, as it is the most interesting topic and unique to Garmin devices. The previous research found multiple vulnerabilities in the third-party apps runtime affecting over a hundred devices [6,8]. Moreover, both of the articles mention that the analysis was far from being comprehensive. Even though Garmin software is a closed source, the researchers were able to find multiple problems in the runtime environment. This suggests it is worth investigating further, as there might be potentially more bugs in the software.

2.2 ATTACK SURFACE

As the focus of the report is third-party apps, the attack surface is limited to the apps and the store. More precisely, for the attack surface, I will focus on the installation of malicious apps. If the adversary is able to install a malicious app on the watch, it is possible to access sensitive user data and possibly tamper with it. There are two stages of the apps that need to be considered:

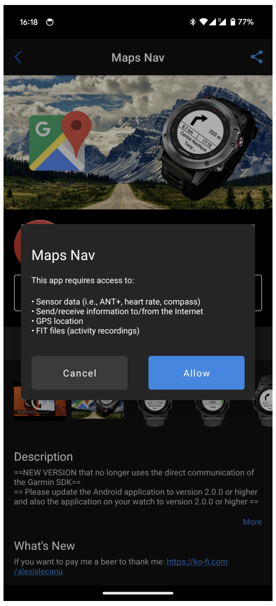
- Installation process
- Runtime

The watch is managed by the phone. Because of that, for the security of the watch, it is important to consider the security of the phone as well. Assuming that the phone itself is secure, the attack surface is limited to the Garmin mobile apps and the communication between the phone and the watch. The installation of the apps is handled by the Connect IQ Store. As such, the security of the store has to be analyzed.

The runtime is handled by the virtual machine, which is responsible for executing the bytecode of the apps. Garmin provides a limited set of APIs that the apps can use. When the user installs the app, they agree to the permissions requested by the app. In this case, it is necessary to analyze if the apps can break the contract and access data or perform actions that they are not supposed to.

3 SECURITY OF CONNECT IQ STORE

3.1 PRELIMINARY ANALYSIS



Maps Nav

This app requires access to:

- Sensor data (i.e., ANT+, heart rate, compass)
- Send/receive information to/from the Internet
- GPS location
- FIT files (activity recordings)

Cancel Allow

Description

==NEW VERSION that no longer uses the direct communication of the Garmin SDK==
== Please update the Android application to version 2.0.0 or higher and also the application on your watch to version 2.0.0 or higher ==

More

What's New

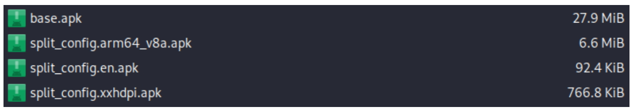
If you want to pay me a beer to thank me: https://ko-fi.com/alexislecanu

Figure 1: Connect IQ - third-party app permissions request

Connect IQ store allows the users to browse and download third-party apps and watch faces to the watch. The applications have descriptions and screenshots that can be added by the developers. Additionally, users can write reviews and rate the apps. Each app has information about the average rating and the number of downloads. Before the app is downloaded, the user is presented with the information about permissions that the app requires, as presented in Figure 1. The store also allows the users to create their own simple watch faces.

Based on the experiments with the watch, it seems that the app is downloaded to the phone and then sent to the watch via Bluetooth.

The analysis of the store does not go very deep, as I decided to focus more on the third-party apps. However, it was necessary to have a basic understanding of the store, especially to analyze the communication between the watch and the phone. This is described in the section 4.8.



base.apk	27.9 MiB
split_config.arm64_v8a.apk	6.6 MiB
split_config.en.apk	92.4 KiB
split_config.xxhdpi.apk	766.8 KiB

Figure 2: Connect IQ - Pulled apk files

3.2 APP DECOMPILATION

I installed the app on the phone and used ADB to pull the apk files, that are presented in Figure 2. The app consists of multiple files, but the main one is `base.apk`. It contains all JVM bytecode, which is the main focus of the analysis. I decompiled the app with the JADX decompiler[1].

The app is obfuscated. It is a good practice as it makes the app size smaller and makes it more difficult to reverse engineer. I was interested to find the code responsible for downloading the app to the phone and sending it to the watch. Searching with several keywords, I was not able to find any code that would be responsible for checking the certificate of the downloaded app. I was looking for names such as the library that was used to sign the app, SHA1, RSA.

Assuming that the app is checking the certificate, it is not necessary for the phone to check it as well. This is way it should be even safer, as the phone would be an additional attack surface. With this conclusion, I decided to experiment later with the installation of an app that was not signed by the store in section 3.7.

3.3 STATIC ANALYSIS

3.4 SNIFFING THE TRAFFIC

To analyze the network security, I tried to sniff the traffic between the phone and Garmin servers. I decided to use mitmproxy[2]. When dynamically analyzing the app, there are two opitons. Either use an emulator or a real device. Emulator usually offers more flexibility and is easier to manipulate. However, it does not have access to Bluetooth, which is necessary to test the app. For this reason, I used a real device. - linux hotspot + mitmproxy - problem with certificates. Two options to solve the problem: - Rooted phone - Modify the app

The minimum supported Android version is 7. Those versions require root access to add certificates supported by applications. Another option is to modify the app to accept user added certificates.

1 https://github.com/skylot/jadx
2 https://mitmproxy.org/

Attempt 1 - Android Unpinner

Use Android Unpinner - [3]

The app starts when connected to mitmproxy. However, when trying to log in, it goes back to the welcome screen. **Not possible to log in**.

Attempt 2 — manually modify the apk and sign again

Using Apktool[4] to unpack the app, modify, and pack again, align zip file and sign with a debug key.

After installing the app, there are some errors with missing resources. I did not investigate it further.

Attempt 3 — only resign the app.

Just resigning the app with a debug key. The app starts, but when trying to log in it seems to work the same as the app from the attempt #1.

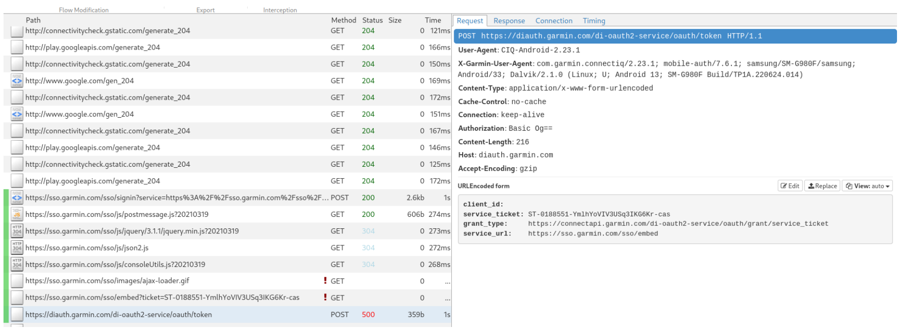
Attempt 4 — use rooted phone

I used my old, no longer used phone — Samsung S20. To root the phone, I followed the instructions found on the XDA Forums[5].

3.5 NETWORK SECURITY,

Based on the traffic that went through mitmproxy:

Used domains: - 'sso.garmin.com' - Qualys SSl Labs analysis: **grade B** - server supports **TLS 1.1** - 'diauth.garmin.com' - Qualys SSl Labs analysis: **grade A**



Path	Method	Status	Size	Time
http://connectivitycheck.gstatic.com/generate_204	GET	204	0	121ms
http://play.googleapis.com/generate_204	GET	204	0	166ms
http://connectivitycheck.gstatic.com/generate_204	GET	204	0	150ms
http://www.google.com/gen_204	GET	204	0	169ms
http://connectivitycheck.gstatic.com/generate_204	GET	204	0	172ms
http://www.google.com/gen_204	GET	204	0	151ms
http://connectivitycheck.gstatic.com/generate_204	GET	204	0	167ms
http://play.googleapis.com/generate_204	GET	204	0	146ms
http://connectivitycheck.gstatic.com/generate_204	GET	204	0	125ms
http://play.googleapis.com/generate_204	GET	204	0	172ms
https://sso.garmin.com/sso/signin?service=https%3A%2F%2Fsso.garmin.com%2Fsso%2F…	POST	200	2.6kb	1s
https://sso.garmin.com/sso/js/postmessage.js?20210319	GET	200	606b	274ms
https://sso.garmin.com/sso/js/jquery/3.1.1/jquery.min.js?20210319	GET	304	0	273ms
https://sso.garmin.com/sso/js/json2.js	GET	304	0	272ms
https://sso.garmin.com/sso/js/consoleUtils.js?20210319	GET	304	0	268ms
https://sso.garmin.com/sso/images/ajax-loader.gif !	GET		0	...
https://sso.garmin.com/sso/embed?ticket=ST-0188551-YmlhYoVIV3USq3IKG6Kr-cas !	GET		0	...
https://diauth.garmin.com/di-oauth2-service/oauth/token	POST	500	359b	1s

Request Response Connection Timing

```
POST https://diauth.garmin.com/di-oauth2-service/oauth/token HTTP/1.1
User-Agent: CIQ-Android-2.23.1
X-Garmin-User-Agent: com.garmin.connectiq/2.23.1; mobile-auth/7.6.1; samsung/SM-G980F/samsung;
Android/33; Dalvik/2.1.0 (Linux; U; Android 13; SM-G980F Build/TP1A.220624.014)
Content-Type: application/x-www-form-urlencoded
Cache-Control: no-cache
Connection: keep-alive
Authorization: Basic Og==
Content-Length: 216
Host: diauth.garmin.com
Accept-Encoding: gzip
```

URLEncoded form Edit Replace View: auto

```
client_id:
service_ticket: ST-0188551-YmlhYoVIV3USq3IKG6Kr-cas
grant_type:     https://connectapi.garmin.com/di-oauth2-service/oauth/grant/service_ticket
service_url:    https://sso.garmin.com/sso/embed
```

Figure 3: Sniffed traffic

3 https://github.com/mitmproxy/android-unpinner
4 https://apktool.org/
5 https://xdaforums.com/t/how-to-exynos-snapdragon-root-s20-series-and-upgrade-firmware.4079353/

The store displays content that can be added by the users. The developer creates a description of the app, screenshots, and other information. Other users can comment on the app. I looked into it, to see how it is handled. If the text was parsed by the app, it could be a potential vector of attack. However, it seems that everything is sent as a plain text, without any parsing on the phone.

3.6 COMMUNICATION BETWEEN THE WATCH AND THE PHONE

I considered analyzing the communication between the watch and the phone. However, it uses Bluetooth LE Secure Connections.

3.7 APP SIDELOADING

- can we install an app that is not signed up by the store? - app signed up only by the developer can be install through the cable, but not through the store - the watch is probably checking the certificate - it suggests that the Bluetooth API for app installation requires the app to be signed by the store - good thing

4 SECURITY OF THIRD-PARTY APPS

4.1 APP RUNTIME

The applications are intended to be run on a wide range of devices, with different hardware configurations. The execution environment needs to be able to enforce access control and isolation. Moreover, the devices are very limited in resources, which makes it difficult to adapt the existing solutions such as Java virtual machine. For those reasons, Garmin decided to create a custom virtual machine and a programming language.

However, creating a correct abstraction layer is a complex task. It creates a large attack surface, and it is challenging to ensure that the implementation is correct.

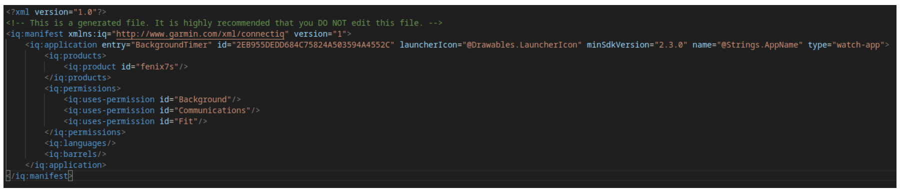
4.2 PRELIMINARY ANALYSIS

The existing research analysed the bytecode of the firmware running on the watch. Unfortunately, the firmware is no longer available for download. There is a chance that it would be possible to reverse engineer the watch hardware and extract the firmware from the device. However, it would require an extensive amount of time and resources and is out of scope of this report. Another option would be to investigate the protocol used to download a new firmware version. It would require additional time and there is no guarantee that it would be possible to download the full firmware.

In the end, I decided to perform the analysis on the simulator provided by Garmin. As it runs directly on the computer, it is possible to perform a more in-depth analysis of the software such as dynamic analysis. The simulator is supposed to guarantee proper isolation, so escaping the sandbox would be a potential vulnerability. Additionally, there is a chance that parts of the code are reused in the firmware running on the watch and the findings would be applicable to the real device.

4.3 PERMISSIONS

Third-party apps can interact with the watch by using the API provided by the SDK. The API is divided into several modules, each of them providing different functionality. Some of those functionalities can be considered sensitive, such as GPS location, heart rate, or access to the internet. Because of that, Garmin implemented a permission



```xml
<?xml version="1.0"?>
<!-- This is a generated file. It is highly recommended that you DO NOT edit this file. -->
<iq:manifest xmlns:iq="http://www.garmin.com/xml/connectiq" version="1">
    <iq:application entry="BackgroundTimer" id="2EB955DEDD684C75824A503594A4552C" launcherIcon="@Drawables.LauncherIcon" minSdkVersion="2.3.0" name="@Strings.AppName" type="watch-app">
        <iq:products>
            <iq:product id="fenix7s"/>
        </iq:products>
        <iq:permissions>
            <iq:uses-permission id="Background"/>
            <iq:uses-permission id="Communications"/>
            <iq:uses-permission id="Fit"/>
        </iq:permissions>
        <iq:languages/>
        <iq:barrels/>
    </iq:application>
</iq:manifest>
```

Figure 4: Garmin third-party app manifest

system, where different permissions grant access to different modules. The app has to define in the manifest file which permissions it requires, and the user has to accept them before installing the app. The manifest file is presented in the figure 4. The system offers some granularity, and can give the user a general overview of what the app can do. However, some, potentially sensitive modules don't require any permissions. For example, the app can access some basic activity data such as heart rate, steps, calories without any permissions. Moreover, in February 2023, Garmin introduced a new module: Complications. It is a publish/subscribe model that allows for the apps to publish different metrics, so that they would be accessible for any watch face. While it makes the access to different metrics more flexible and accessible, it also removes the granularity of the permissions. Every app can potentially publish very sensitive data, without the user being aware of it. Then every watch face can access all this data if it requests 'Complications' permission.

4.4 BUILDING PROCESS

In order to understand the runtime environment, I started with the analysis of the building process. As it was mentioned, it is recommended to use Visual Studio Code with the extension provided by Garmin. During the compilation, the code is translated to mid-level intermediate representation (MIR). The files containing the representations could be used for better understanding of the final artefact.

(TODO: elaborate that a PRG file is created)

Visual Studio calls the compiler, which is a jar file called *monkeybrains.jar*. The compiler is written in Java, which makes the analysis of the code relatively straightforward. Nonetheless, it is a complicated process consisting of several stages. For the analysis, I decided to use the JADX decompiler[1].

After loading the jar file, I noticed that a large part of the relevant code is not obfuscated. Then I proceeded to analyse the code. The following classes are of particular interest:

- `com.garmin.monkeybrains.compiler2.Compiler2` — main class responsible for the compilation process.

1 https://github.com/skylot/jadx

- `com.garmin.connectiq.common.monkeyc.AssemblerOpcode` — contains a list of instructions in the final bytecode.
- `com.garmin.connectiq.common.signing.SigningUtils` — class responsible for signing the app, described in more detail in the section 4.5.

Additionally, the SDK provides a mapping of all API methods to the number values, which is probably some type of address.(TODO: improve)

I was able to find the code responsible for signing the application. It is described in more detail in the section 4.5.

4.5 SIGNING

Garmin requires that the apps are signed to be installed on the watch [3]. The app has to be signed by the developer with their private key. In order for the app to be updated, it has to be signed with the same key. When the store has approved the app, it is additionally signed wit the store signature.

The described process enforces that the app can be updated only by the developer having access to the original private key. Additionally, when the app is downloaded from the store, the watch can verify that it has been approved by the store.

Based on the analysis of the compiler, class `SigningUtils` is responsible for signing the app. It uses `java.security.Signature` class to generate the signature. The signing algorithm uses SHA1 hash of the bytecode that is signed with a private RSA key. The key is 4096 long, which is more than recommended. For the signing, conventions described in PKCS #1 v2.2 are used [4, 5]. The pseudocode of the signing algorithm is presented in the listing 1. The signature computed by the store is generated in a similar way, does not include the modulus and exponent, and uses a different magic number. The store probably uses the public key in the developer signature to verify and save it after the first upload of the app.

```
def sign_with_developer_signature():
    app_bytes = read_prg_file()
    app_bytes, terminator = remove_terminator(app_bytes)
    signature, modulus, exponent = compute_signature(
        app_bytes)

    dev_signature = byte_array()
    dev_signature.append(magic_number)
    dev_signature.append(signature_length)
    dev_signature.append(signature)
    dev_signature.append(modulus)
    dev_signature.append(exponent)

```

```
        signed_app = byte_array()
        signed_app.append_all(app_bytes, dev_signature,
            terminator)
        write_prg_file(signed_app)
```

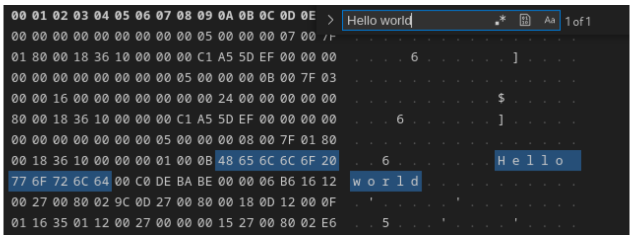
Listing 1: Pseudocode of the signing algorithm, developer signature

SHA-1 is no longer considered secure against well-funded opponents. NIST formally deprecated use of this hash function in 2011. In 2020 there was a paper published demonstrating a chosen-prefix collision attack. A potential vector of attack would be to create a malicious app with the same hash as the original app. At the current state of the art, there is no viable solution to find a collision to a given hash for a chosen prefix. However, the function has been already broken and in the upcoming years new attacks might be discovered.

4.6 MODIFYING THE EXECUTABLE

To perform dynamic analysis of the virtual machine, it is useful to be able to edit the executable. However, after changing the bytecode, it is necessary to sign the executable again. With the knowledge gained from the previous section, I created a Kotlin script that signs the app again.

The code follows the same instructions as the original method in listing 1. The only difference is that the old signature is replaced with the new one.



```
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E    > Hello world    1 of 1
00 00 00 00 00 00 00 00 00 05 00 00 00 07 00 7F  ................
01 80 00 18 36 10 00 00 00 C1 A5 5D EF 00 00 00  ....6......]....
00 00 00 00 00 00 00 00 05 00 00 00 0B 00 7F 03  ................
00 00 16 00 00 00 00 00 00 00 24 00 00 00 00 00  ..........$.....
80 00 18 36 10 00 00 00 C1 A5 5D EF 00 00 00 00  ...6......].....
00 00 00 00 00 00 00 05 00 00 00 08 00 7F 01 80  ................
00 18 36 10 00 00 00 01 00 0B 48 65 6C 6C 6F 20  ..6.......Hello 
77 6F 72 6C 64 00 C0 DE BA BE 00 00 06 B6 16 12  world...........
00 27 00 80 02 9C 0D 27 00 80 00 18 0D 12 00 0F  .'.....'........
01 16 35 01 12 00 27 00 00 00 15 27 00 80 02 E6  ..5...'......'..
```

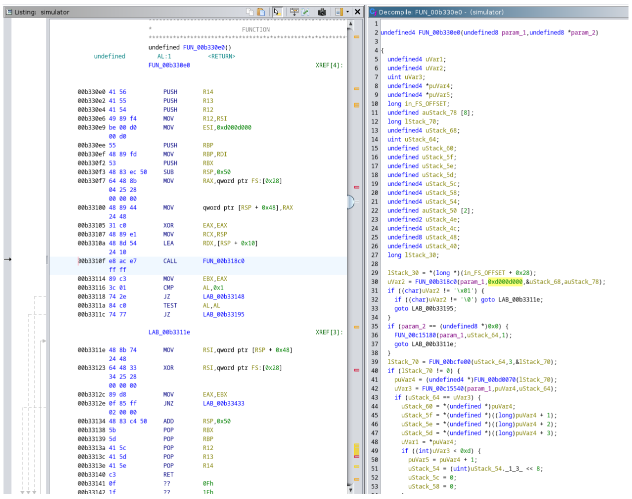
Figure 5: Prg file in hex editor

To verify that the script works correctly, I created a simple app that shows a message 'Hello world'. Then I opened the PRG file in a hex editor and searched for the message, as presented in the figure 5. Consequently, I modified the string and signed the app with the created script. After installing the app on the watch and the simulator, the new message was displayed. When trying to install the app without the new signature, the watch and the simulator would display an error message.

4.7 SIMULATOR

Garmin SDK provides a simulator for testing the apps. It is used by the Visual Studio Code extension to run the apps. I found the executable in the SDK bin folder. The file starts with ELF header, which means that it is a binary executable file. To analyse bytecode, I decided to use Ghidra[2]. It is free and open-source software for reverse engineering.

After loading the file, a list of assembly instructions is displayed. Fortunately, Ghidra has an option to analyse and decompile the code to C.



```
                        *                 FUNCTION
                        ****************************************************************
                        undefined FUN_00b330e0()
       undefined          AL:1         <RETURN>
                        FUN_00b330e0                                            XREF[4]:

   00b330e0 41 56         PUSH      R14
   00b330e2 41 55         PUSH      R13
   00b330e4 41 54         PUSH      R12
   00b330e6 49 89 f4      MOV       R12,RSI
   00b330e9 be 00 d0      MOV       ESI,0xd000d000
            00 d0
   00b330ee 55            PUSH      RBP
   00b330ef 48 89 fd      MOV       RBP,RDI
   00b330f2 53            PUSH      RBX
   00b330f3 48 83 ec 50   SUB       RSP,0x50
   00b330f7 64 48 8b      MOV       RAX,qword ptr FS:[0x28]
            04 25 28
            00 00 00
   00b33100 48 89 44      MOV       qword ptr [RSP + 0x48],RAX
            24 48
   00b33105 31 c0         XOR       EAX,EAX
   00b33107 48 89 e1      MOV       RCX,RSP
   00b3310a 48 8d 54      LEA       RDX,[RSP + 0x10]
            24 10
   00b3310f e8 ac e7      CALL      FUN_00b318c0
            ff ff
   00b33114 89 c3         MOV       EBX,EAX
   00b33116 3c 01         CMP       AL,0x1
   00b33118 74 2e         JZ        LAB_00b33148
   00b3311a 84 c0         TEST      AL,AL
   00b3311c 74 77         JZ        LAB_00b33195

                        LAB_00b3311e                                            XREF[3]:

   00b3311e 48 8b 74      MOV       RSI,qword ptr [RSP + 0x48]
            24 48
   00b33123 64 48 33      XOR       RSI,qword ptr FS:[0x28]
            34 25 28
            00 00 00
   00b3312c 89 d8         MOV       EAX,EBX
   00b3312e 0f 85 ff      JNZ       LAB_00b33433
            02 00 00
   00b33134 48 83 c4 50   ADD       RSP,0x50
   00b33138 5b            POP       RBX
   00b33139 5d            POP       RBP
   00b3313a 41 5c         POP       R12
   00b3313c 41 5d         POP       R13
   00b3313e 41 5e         POP       R14
   00b33140 c3            RET
   00b33141 0f            ??        0Fh
   00b33142 1f            ??        1Fh
```

```c

undefined4 FUN_00b330e0(undefined8 param_1,undefined8 *param_2)

{
  undefined4 uVar1;
  undefined4 uVar2;
  uint uVar3;
  undefined4 *puVar4;
  undefined4 *puVar5;
  long in_FS_OFFSET;
  undefined auStack_78 [8];
  long lStack_70;
  undefined4 uStack_68;
  uint uStack_64;
  undefined uStack_60;
  undefined uStack_5f;
  undefined uStack_5e;
  undefined uStack_5d;
  undefined4 uStack_5c;
  undefined4 uStack_58;
  undefined4 uStack_54;
  undefined auStack_50 [2];
  undefined2 uStack_4e;
  undefined4 uStack_4c;
  undefined8 uStack_48;
  undefined4 uStack_40;
  long lStack_30;

  lStack_30 = *(long *)(in_FS_OFFSET + 0x28);
  uVar2 = FUN_00b318c0(param_1,0xd000d000,&uStack_68,auStack_78);
  if ((char)uVar2 != '\x01') {
    if ((char)uVar2 != '\0') goto LAB_00b3311e;
    goto LAB_00b33195;
  }
  if (param_2 == (undefined8 *)0x0) {
    FUN_00c15180(param_1,uStack_64,1);
    goto LAB_00b3311e;
  }
  lStack_70 = FUN_00bcfe00(uStack_64,3,&lStack_70);
  if (lStack_70 != 0) {
    puVar4 = (undefined4 *)FUN_00bd0070(lStack_70);
    uVar3 = FUN_00c15540(param_1,puVar4,uStack_64);
    if (uStack_64 == uVar3) {
      uStack_60 = *(undefined *)puVar4;
      uStack_5f = *(undefined *)((long)puVar4 + 1);
      uStack_5e = *(undefined *)((long)puVar4 + 2);
      uStack_5d = *(undefined *)((long)puVar4 + 3);
      uVar1 = *puVar4;
      if ((int)uVar3 < 0xd) {
        puVar5 = puVar4 + 1;
        uStack_54 = (uint)uStack_54._1_3_ << 8;
        uStack_5c = 0;
        uStack_58 = 0;
```

Figure 6: Simulator decompilation

The header of the third-party apps is tagged with value '0xD000D000' [6]. Searching for it in the simulator binary reveals two references.

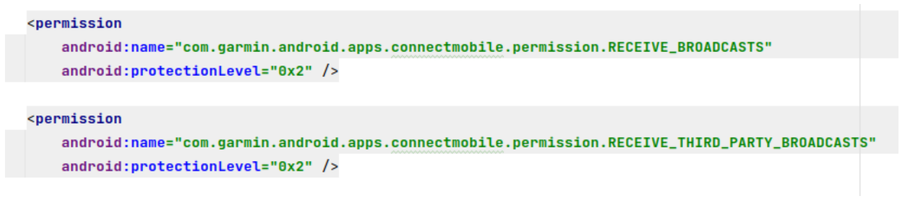
From this point, it would be theoretically possible to analyse the code and reverse engineer the virtual machine. However, doing it without any documentation, where the names of the functions and variables are not known would be very challenging. As it is not possible to do it in a reasonable amount of time, I decided to focus on fuzzing the virtual machine. This is described in the section 5.

4.8 COMMUNICATION BETWEEN THE WATCH AND THE PHONE

Garmin has multiple apps for managing the watch and the data. As such, it is interesting to analyse the communication between them and the watch. I analyzed the Manifest files in Connect IQ Store and Garmin Connect apps. Both of them use Bluetooth permissions to connect with the watch. Additionally, they define Garmin custom permissions as can be seen in the figure 7. The permissions have set protection level to 'signature'(0x2), which means that only apps signed with the same key can use them. They are used to be able to receive

2 https://ghidra-sre.org/

specific broadcasts, probably to communicate with themselves. It is a functionality provided by Android and should offer an appropriate level of security.



```
<permission
    android:name="com.garmin.android.apps.connectmobile.permission.RECEIVE_BROADCASTS"
    android:protectionLevel="0x2" />

<permission
    android:name="com.garmin.android.apps.connectmobile.permission.RECEIVE_THIRD_PARTY_BROADCASTS"
    android:protectionLevel="0x2" />
```

Figure 7: Declared permissions by Connect IQ Store

When it comes to third-party apps, they can communicate with the phone by using the API provided by the SDK. Garmin created SDK for Android and iOS that allows to create a companion mobile app for the watch app. They created an example of watch and Android app that communicate with each other[3]. The Android SDK uses Broadcasts under the hood to communicate with Garmin Connect app. It allows the programmer to query for available Garmin devices connected to the phone. Then it is possible to establish communication with the chosen app on the watch.

The first thing that can be noticed is that the SDK is able to get a list of all Garmin devices connected to the phone without any permissions. They use Broadcasts that are available to all apps. This creates a potential privacy issue, as any app can get the information if the user has a Garmin device connected to the phone. Moreover, it is possible to get the information about the device, such as the model name.

Another issue is that the communication between the app on the watch and the companion app is not encrypted. The SDK offers a function to register a listener for events from the watch app, given the app ID. However, when looking at the implementation, it is a wrapper around the Broadcasts. It is possible to register a listener for all events from all apps. This means that any app on the phone can listen to the messages from any watch app. I did not find any mention that would warn the developers about this issue.

Following this finding, I decided to test this hypothesis. I created a simple watch app that sends a message to the companion app. Then, based on the SDK, I created an Android app that listens to all messages from all watch apps. When I installed both apps, the Android app was able to receive the messages from the watch app with the information about the origin.

TODO: add screenshot of the app

Having this information, I proceeded to investigate if it might be an issue for existing apps in the store. Firstly, I looked into Spotify and Komoot apps. Both of them are very popular and have a large number

3 https://github.com/garmin/connectiq-android-sdk

of downloads. They have a full version of the app available on the phone and a companion app on the watch. However, they do not use the SDK provided by Garmin. Instead, they connect to their servers directly. It might be because it was easier to implement, or because they wanted to have more control over the communication.

Then I tried to find apps that use the SDK provided by Garmin. I found a developer 'r.485' that has published multiple apps in the store with the companion app on the phone. However, when I installed the apps, they did not work correctly.

At this point, I decided to stop the investigation. While it is a privacy issue, I could not find any apps that it would affect.

As Garmin already uses custom permissions for communication between their apps, I wondered why they did not use it for the SDK. According to Android documentation, [1] it is possible to send broadcasts that can be only received by apps with specific permissions. It is also possible to define a custom permission that can be requested by other apps. However, custom permissions have to be already defined when such an app is being installed. This is not always the case, and because of that, it is probably not a viable option for Garmin.

5 FUZZING THIRD-PARTY APPS

5.1 INTRODUCTION

Static analysis of assembly code is a very time-consuming process that requires extensive knowledge. Without any documentation, source code provided by the vendor, or any other help, it is extremely difficult to understand the code.

For this reason, I decided to use dynamic analysis to find vulnerabilities in the code. As I am focusing on analysis of the virtual machine in the simulator, there are more possibilities to test the behaviour during runtime. The simulator allows running the code in a controlled environment, which makes it easier to test different scenarios. With this approach, there is no need to understand the code. It is also valuable when the code base is large and complex. Additionally, instead of spending time on reverse engineering, it is possible to write a fuzzer that will do the difficult work. Sometimes it might be necessary to run the fuzzer for a long time. This, however, does not require any human interaction and can be scaled to run on multiple machines or threads. The expected outcome of the fuzzer is a binary file that crashes the virtual machine and can be run on the watch.

Finding vulnerabilities in the simulator is aimed at achieving two potential objectives. Firstly, it might be possible to escape the sandbox and execute arbitrary code on the host machine. Secondly, vulnerabilities in the simulator might be present in the real device as well.

5.2 ENVIRONMENT SETUP

When fuzzing a program, the most basic configuration would be to have a fuzzer and provide it with a program and the starting input. Then the fuzzer would run the program with different modifications of the input and check if it crashes.

However, in this case, the setup is more complicated. Simulator is run a standalone program. Then the application has to be loaded with an additional command line script. As such, the fuzzer needs to be aware of both tools.

Moreover, when modifying the input, the virtual machine requires it to be signed. This means that to test the execution after each modification, the input needs to be signed again.

5.3 FUZZING SOLUTIONS

One of the most popular fuzzing solutions is AFL++ [11]. It is an improved fork to Google's AFL[1], which is no longer maintained. AFL++ is an open-source tool for fuzzing a wide range of software. It is coupled with instrumentation-guided genetic algorithm, having a possibility to be extended for different use cases.

(TODO: explain that I decided to write my own)

5.4 FUZZING PROCESS

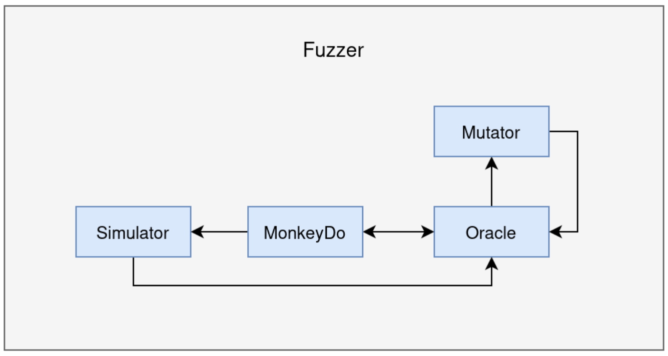


Figure 8: Fuzzing process

The fuzzer consists of four main components: Mutator, Simulator, MonkeyDo, and Oracle, as presented in Figure 8. The fuzzing is performed in an iterative manner. The Mutator generates a new input, then Oracle tests the execution of the input. Based on the result, the Mutator either mutates the current input or reverts the previous mutation. This way, the input oscillates closely between two states: one that barely works and another one that does not run successfully. With this continuous modification of the input, the fuzzer explores different variations of the bytecode until it finds one that is not handled correctly by the virtual machine. To make it possible to reproduce the steps, the fuzzer uses a random generator with a fixed seed, defined at the beginning of the script.

Mutator

```
originalApp = load("seed_app.prg")
currentApp = originalApp
lastMutation = emptyList()

def mutate(passed: Boolean):
```

1 https://github.com/google/AFL

```
    if passed:
        mutateCurrentApp()
    else:
        revertMutations()
    return sign(currentApp)

def mutateCurrentApp():
    mutation = generateMutation()
    for byte in mutation:
        currentApp[byte] = flipRandomBit(currentApp[byte])
    lastMutation = mutation

def revertMutations():
    lastMutation, bytesToRevert = splitInHalf(lastMutation)
    for byte in bytesToRevert:
        currentApp[byte] = originalApp[byte]

def generateMutation():
    mutationSize = mutationRate*len(currentApp)
    return [randomByte() for i in range(0, mutationSize)]
```

Listing 2: Pseudocode of the mutating algorithm

Mutator is responsible for generating a new input for the simulator. It does that by mutating the original(seed) application and signing it again as described in section 4.6. The pseudocode of the mutating algorithm is presented in Listing 2. Depending on the result of the previous execution, the mutator either mutates the current application or reverts the previous mutation. To make the process more efficient, the mutator does not revert the whole mutation, but only half of it. Using the divide-and-conquer approach, it is possible to find the byte that broke the application faster.

Simulator

The simulator component is responsible for running the simulator executable. In Figure 9, the simulator is running the seed application. The component monitors the output of the simulator, which can be used by the Oracle to decide if provided input has been executed successfully. This information is needed to decide if the mutator should mutate the current application or revert the previous mutation. Example output of the simulator is presented in Figure 10. Additionally, the component detects when the simulator crashes. This information is used by the Oracle component to determine if a bug has been found.

MonkeyDo

Oracle component is using MonkeyDo script to load the application into the simulator. The script outputs logs from the loaded application.

It also does a partial parsing of the application and might sometimes output errors.

Oracle

The Oracle component tests the execution of the application generated by the Mutator. It uses the output of the Simulator and MonkeyDo to determine if the execution was successful. At this stage, it was important to investigate possible outputs of the MonkeyDo script and the simulator. Based on those outputs, a decision is made if the application should be mutated or reverted.

The app is programmed to print a message `Okay` when it is executed. This message is used to determine if the execution was successful. However, it may take a few seconds for the message to appear. To speed up the process, the Oracle also checks if any of the expected error messages has been printed. If it is the case, the execution is considered to be unsuccessful.

Sometimes when the application is modified, the simulator will output an error message that signature check failed. The message contains the name of the app file. When this happens, the application is considered invalid and the mutation is reverted. However, sometimes the same output is produced again when the next application is loaded. Because of that, every version of the application has to have a different file name. This way, it can be determined if the error is caused by the previous application or the current one.

5.5 FUZZING RESULTS

For the fuzzing, I created a simple application that prints a message *Okay* when executed. The application is based on an example provided by the SDK, called *Confirmation Dialog*.

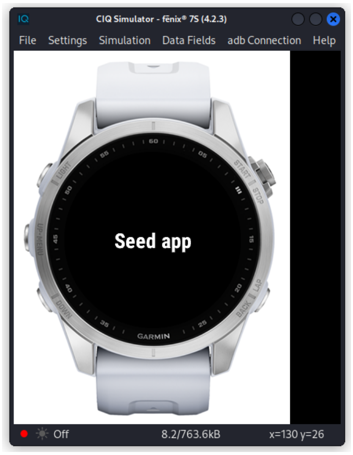
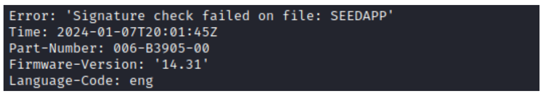


Figure 9: Simulator running seed app

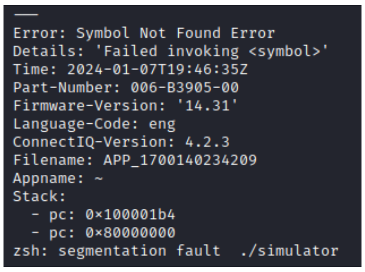


```
Error: 'Signature check failed on file: SEEDAPP'
Time: 2024-01-07T20:01:45Z
Part-Number: 006-B3905-00
Firmware-Version: '14.31'
Language-Code: eng
```

Figure 10: Simulator output when signature verification fails

I was running the fuzzer with at most 50 iterations, so that it would be possible to easily reproduce all steps. Each run of the fuzzer started with a different seed for a random generator.

After the first run, I noticed that sometimes there would be an error message: *Permissions required*. Probably some modifications caused the application to access parts of the SDK that it did not have permission to. After this finding, I decided to modify the application to request all permissions. This way, it would be possible to detect potential bugs in all parts of the SDK.



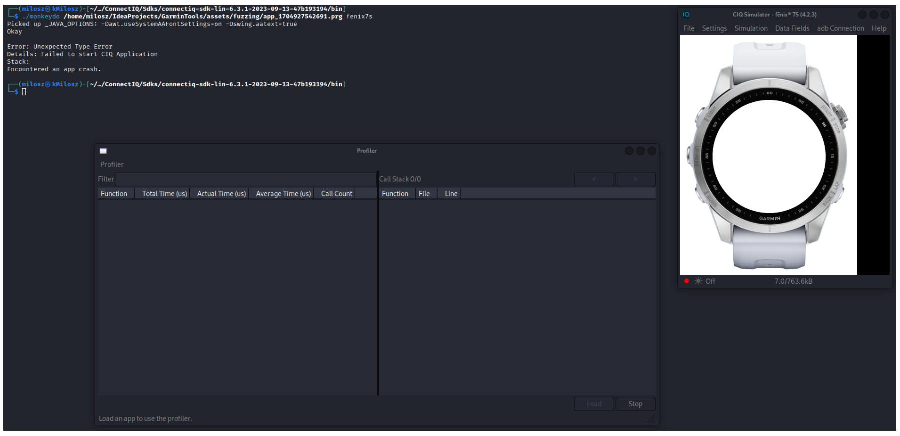
```
---
Error: Symbol Not Found Error
Details: 'Failed invoking <symbol>'
Time: 2024-01-07T19:46:35Z
Part-Number: 006-B3905-00
Firmware-Version: '14.31'
Language-Code: eng
ConnectIQ-Version: 4.2.3
Filename: APP_1700140234209
Appname: ~
Stack:
  - pc: 0x100001b4
  - pc: 0x80000000
zsh: segmentation fault  ./simulator
```

Figure 11: Simulator crash

With this configuration, the fuzzer was able to consistently find an input that was crashing the simulator. Roughly every second or third run of the fuzzer a new such input was found. The crash was caused by a segmentation fault, as presented in Figure 11. This is a bug in the simulator, as it should not crash when executing an application. However, it is challenging to assess if this bug can be exploited. Potentially, it might be possible to execute arbitrary code on the host machine. However, it would require a further investigation into the machine code of the simulator.

Additionally, I tried to fuzz the application without any permissions, to compare the results. It seemed that the crashes were happening more rarely. However, during one of the runs, I noticed a strange behaviour of the simulator. Once, when the application was executed, the simulator started profiling and opened a window with the profiler, as presented in Figure 12. The application was executed, printed the message *Okay*, and then crashed. I can see two possible explanations for this behaviour. Firstly, probably the most likely, there is a bug during the initial parsing/loading of the application that triggers the profiler. Or maybe a no documented feature of the simulator. Secondly, there is a chance that the application started the profiler from the virtual machine.

(TODO: write about testing those apps on the watch)



```
┌──(milosz㉿kMilosz)-[~/…/ConnectIQ/Sdks/connectiq-sdk-lin-6.3.1-2023-09-13-47b193194/bin]
└─$ ./monkeydo /home/milosz/IdeaProjects/GarminTools/assets/fuzzing/app_1704927542691.prg fenix7s
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
Okay

Error: Unexpected Type Error
Details: Failed to start CIQ Application
Stack:
Encountered an app crash.

┌──(milosz㉿kMilosz)-[~/…/ConnectIQ/Sdks/connectiq-sdk-lin-6.3.1-2023-09-13-47b193194/bin]
└─$ 
```

Figure 12: Weird simulator behaviour

Those results show that there are bugs in the simulator. This opens an opportunity to continue the investigation to assess if it is possible to find any vulnerabilities that can be exploited.

6 PRIVACY

7 CONCLUSION

7.1 SUMMARY

7.2 FUTURE WORK

- Investigation into the findings from the fuzzy testing
- Further analysis of the mobile Garmin apps
- Reverse engineering of the firmware/virtual machine

BIBLIOGRAPHY

[1] Broadcasts overview. https://developer.android.com/guide/components/broadcasts.

[2] Garmin connect iq devices. https://developer.garmin.com/connect-iq/compatible-devices/.

[3] Garmin security. https://developer.garmin.com/connect-iq/core-topics/security/.

[4] Java signature algorithms. https://docs.oracle.com/en/java/javase/17/docs/specs/security/standard-names.html#signature-algorithms.

[5] Pkcs #1: Rsa cryptography specifications version 2.2. https://datatracker.ietf.org/doc/html/rfc8017.

[6] A watch, a virtual machine, and broken abstractions. https://www.atredis.com/blog/2020/11/4/garmin-forerunner-235-dion-blazakis, 2020.

[7] Ethical hacking of garmin's sports watch. http://kth.diva-portal.org/smash/get/diva2:1612537/FULLTEXT01.pdf, 2021.

[8] Compromising garmin's sport watches: A deep dive into garminos and its monkeyc virtual machine. https://www.anvilsecure.com/blog/compromising-garmins-sport-watches-a-deep-dive-into-garminos-and-its-monkeyc-
html, 2022.

[9] Garmin gcd firmware update file format. https://www.memotech.franken.de/FileFormats/Garmin_GCD_Format.pdf, 2022.

[10] A cybersecurity audit of the garmin venu. http://kth.diva-portal.org/smash/get/diva2:1757507/FULLTEXT01.pdf, 2023.

[11] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ : Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020.