# Security Analysis of Digital Post Application

Miłosz Głowaczewski

*Abstract*— **During the analysis the following vulnerabiliteis have been found. Garmin Android apps broadcast the model of the watch connected. Every Android app can listen to that. Another finding is that Garmin offers an API to communicate third party watch apps with their companion phone apps. The traffic is in plain text and broadcasted to every installed Android app. During fuzzing it was possible to crash a simulator with a modified prg file. This might also mean that remote code execution attack is possible on a real watch.**

## I. Introduction

### A. 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 the health and fitness tracking it also means that it collects more sensitvie data about the users.

Garmin fitness watches can accompany the users day and night, being able to last even a week on a single charge.They offer a wide range of sensors such as: heart rate, pulse oximeter, temperature, gps, compass, accelerometer, gyroscope, altimeter and barometer, light sensor. This allows to collect and callculate an enormous amount of different sensitive metrics. As a result, the potential consequences of unauthorized access to the users data are severe.

The watches can work indepedently, collecting and processing the data offline. Hovewer 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 wawtch and the data. This means that the data is send over a Bluetooth protocol from the watch to the phone. Additionally Garmin recomends 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 develoers to create apps and publish them in a Gramin store. It makes the watch extensible and more universal, but it also alows for a third party developer to gain access to some of the user data.

The described software offers a powerfull toolset for the analysis of fitness data for every user. However it also create a large amount of possible attack surfaces. It is also important to notice that the Garmin software is not open source and there is not bounty program for finding voulnerabilities.

In this report, a broad analysis of diffrent vector attacks has been performed.

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

- Garmin uses deprecatred 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 the their companion apps are in plain text, available to all apps installed on the Andoid phone
- Permission system of the third party watch apps is not as well seperated into different categories as it is in Android, iOS phones.
- Fuzzy testing suggetst that it may be possible to escape the virtual machine, that is running the third party apps.

### B. Software description

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

Garmin created SDK for third party app development. It is integrated with Visual Studio Code through the Monkey C extension. There is an official documentation describing the development process with Visual Studio Code. However, it doesn't describe the specific scripts and tools contained in the SDK that are used by the extension. The SDK consists mostly of Java class used for compiling and packaging the application. Additionally, a simulator is provided for testing the apps. Unfortunately, it is implemented as a binary ELF file, which makes the reverse engineering more difficult. Moreover, other scripts are provided for other tasks such as uploading an app to the simulator.

List of tools:

- monkeybrains.jar - compiles and build the project
- simulator - simulates a chosen Garmin device
- monkeydo - executes Connect IQ executable on the simulator
- Other scripts: barrelbuild, barreltest, connectiq, era, mdd, monkeyc, monkeygraph, monkeydoc, shell, monkeymotion

## II. Related work

## III. Threat analysis

In this report the folling attack surfaces will be analyzed:

- Third party apps
  - Test

### A. Test environment

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

## IV. Security of Connect IQ store

### A. Preliminary analysis

Application allows the users to download third party apps to the watch. The user can select an app to install it.

Notes:

- the phone downloads the app
- sends it to the watch via BT

### B. App decompilation

I decompiled the app with JADX. The app is obfuscated.

I couldn't find any code that would be responsible for checking the certificate of the downloaded app. I was looking for keywords such as the library that was used during a build process, SHA1, RSA.

Probably there is not much of a point to check the certificate on the phone, assuming that the watch is doing it.

I didn't manage to find the code responsible for downloading the app.

### C. Static analysis

### D. Network security

- emulator does not have access to BT - 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) Attempt 1 - Android Unpinner:* Use Android Unpinner - https://github.com/mitmproxy/android-unpinner

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**.

*2) Attempt 2 — manually modify the apk and sign again:* Using Apktool(https://apktool.org/) 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.

*3) Attempt 3 - just 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.

*4) Attempt 4 — use rooted phone:* Short description about the root

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**

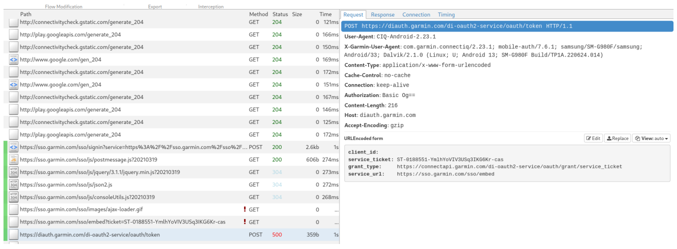
## V. Security of third party apps

### A. Permissions

### B. App runtime

### C. Simulator

Searching for value ‘0xD000D000‘ reveals two references in the code. Based on the

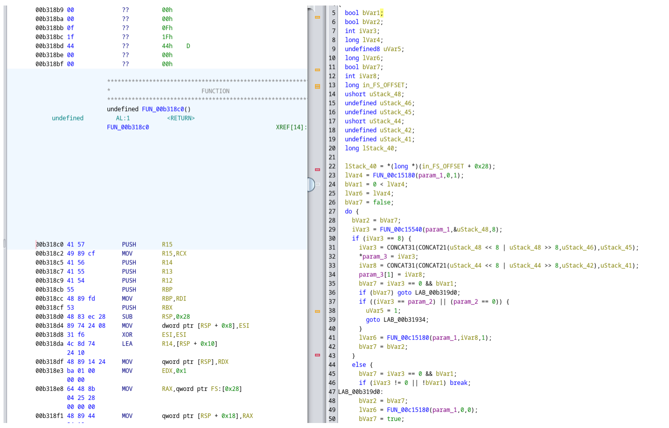


| | Path | Method | Status | Size | Time |
|---|---|---|---|---|---|
| | http://connectivitycheck.gstatic.com/generate_204 | GET | 204 | 0 | 121ms |
| | http://play.googleapis.com/generate_204 | GET | 204 | 0 | 166ms |
| | http://connectivitycheck.gstatic.com/generate_204 | GET | 204 | 0 | 150ms |
| | 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 |

```
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
```

```
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
```

Fig. 1: Sniffed traffic



```
00b318b9 00          ??          00h
00b318ba 00          ??          00h
00b318bb 0f          ??          0Fh
00b318bc 1f          ??          1Fh
00b318bd 44          ??          44h    D
00b318be 00          ??          00h
00b318bf 00          ??          00h

                     *************************************************************
                     *                         FUNCTION
                     *************************************************************
                     undefined FUN_00b318c0()
      undefined        AL:1          <RETURN>
                     FUN_00b318c0                                         XREF[14]:

00b318c0 41 57       PUSH        R15
00b318c2 49 89 cf    MOV         R15,RCX
00b318c5 41 56       PUSH        R14
00b318c7 41 55       PUSH        R13
00b318c9 41 54       PUSH        R12
00b318cb 55          PUSH        RBP
00b318cc 48 89 fd    MOV         RBP,RDI
00b318cf 53          PUSH        RBX
00b318d0 48 83 ec 28 SUB         RSP,0x28
00b318d4 89 74 24 08 MOV         dword ptr [RSP + 0x8],ESI
00b318d8 31 f6       XOR         ESI,ESI
00b318da 4c 8d 74    LEA         R14,[RSP + 0x10]
         24 10
00b318df 48 89 14 24 MOV         qword ptr [RSP],RDX
00b318e3 ba 01 00    MOV         EDX,0x1
         00 00
00b318e8 64 48 8b    MOV         RAX,qword ptr FS:[0x28]
         04 25 28
         00 00 00
00b318f1 48 89 44    MOV         qword ptr [RSP + 0x18],RAX
```

```
  bool bVar1;
  bool bVar2;
  int iVar3;
  long lVar4;
  undefined8 uVar5;
  long lVar6;
  bool bVar7;
  int iVar8;
  long in_FS_OFFSET;
  ushort uStack_48;
  undefined uStack_46;
  undefined uStack_45;
  ushort uStack_44;
  undefined uStack_42;
  undefined uStack_41;
  long lStack_40;

  lStack_40 = *(long *)(in_FS_OFFSET + 0x28);
  lVar4 = FUN_00c15180(param_1,0,1);
  bVar1 = 0 < lVar4;
  lVar6 = lVar4;
  bVar7 = false;
  do {
    bVar2 = bVar7;
    iVar3 = FUN_00c15540(param_1,&uStack_48,8);
    if (iVar3 == 8) {
      iVar3 = CONCAT31(CONCAT21(uStack_48 << 8 | uStack_48 >> 8,uStack_46),uStack_45);
      *param_3 = iVar3;
      iVar8 = CONCAT31(CONCAT21(uStack_44 << 8 | uStack_44 >> 8,uStack_42),uStack_41);
      param_3[1] = iVar8;
      bVar7 = iVar3 == 0 && bVar1;
      if (bVar7) goto LAB_00b319d0;
      if ((iVar3 == param_2) || (param_2 == 0)) {
        uVar5 = 1;
        goto LAB_00b31934;
      }
      lVar6 = FUN_00c15180(param_1,iVar8,1);
      bVar7 = bVar2;
    }
    else {
      bVar7 = iVar3 == 0 && bVar1;
      if (iVar3 != 0 || !bVar1) break;
LAB_00b319d0:
      bVar2 = bVar7;
      lVar6 = FUN_00c15180(param_1,0,0);
      bVar7 = true;
```

Fig. 2: Simulator decompilation

[article](https://www.atredis.com/blog/2020/11/4/garmin-forerunner-235-dion-blazakis) it is a PRG header tag. The code doesn't look similar to the one included in the article. Theoretically, I could try to analyze the logic for parsing the PRG file. However, it doesn't seem like something that can be done in a reasonable amount of time.

### D. Building process

The compiler is written in Java, which makes the analysis of the code relatively simple. Nonetheless, the compilation is a complicated process consisting of several stages. Java class ‘Opcode‘ contains a list of instructions in the final bytecode. Additionally, SDK provides a mapping of all API methods to the number values, which is probably some type of address.

During the compilation, the code is translated to mid-level intermediate representation (MIR). The files containing the representations are created during the build process and can be used for better understanding of the final artifact.

### E. Bytecode analysis

### F. Modifying the executable

In order to test the security 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. After analysis of *monkeybrains.jar* file, I created a kotlin script that signs the app again.

### G. Signing

The generated app bytecode has to be signed by the developer. 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.

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. It still doesn't offer a 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.

Algorithm for signing: - read bytes from PRG file that hasn't been signed yet - don't include the bytes at the end of the file (called TERMINATOR in the code) - compute the signature with Java security signature library (SHA1withRSA) - Append the file bytes Developer signature, consisting of: - magic number - length of the whole signature - signature - modulus - exponent - Or in the case of the store signature: - a different magic number - length of the whole signature - signature - Append the terminating bytes that were skipped before

### *H. Communication between the watch and the phone*

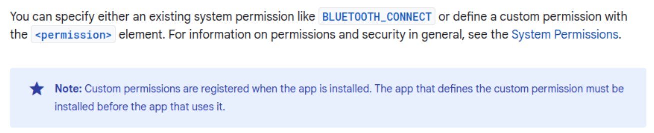
*1) Garmin communication between the Android apps:* Garmin apps define their own permissions. One of them is for example `com.garmin.android.apps.connectmobile.permission.RECEIVE_BROADCASTS`, which allows to listen for action `ACTION_ON_FIND_MY_PHONE_MESSAGE_RECEIVED`, which triggers sound on the phone. I don't know, however, who sends this action.

**Privacy issue 1** - Every app can register for garmin ConnectIQ service without registering any permissions. It can give the information if and what Garmin device the user has connected to the phone.

It seems like any phone app can receive events from all watch apps. There is no authentication.

I have a proof of concept, but idk which apps are using it, if any...

According to Android documentation: [About broadcasts](https://developer.android.com/guide/components/broadcasts), 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. However, custom permissions have to be already defined when such app is being installed. Because of that, it is probably not a viable option for Garmin.



You can specify either an existing system permission like `BLUETOOTH_CONNECT` or define a custom permission with the `<permission>` element. For information on permissions and security in general, see the System Permissions.

★ **Note:** Custom permissions are registered when the app is installed. The app that defines the custom permission must be installed before the app that uses it.

Fig. 3: Android custom permissions note

**Privacy issue 2** - Third party apps can user's personal information or access the internet. To do so, they need to request permission, specific to the used module. For example, 'Communications', 'Fit', 'Positioning'. This system offers some granularity, and the user can see what data will the app have access to. However, 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. However, it should be considered that it removes the granularity of permissions. Every app can potentially publish very sensitive data. And every watch face can access all this data if it requests 'Complications' permission.

### *I. Sandboxing*

### *J. App sideloading*

## VI. Fuzzing third party apps

### *A. Environment setup*

### *B. Fuzzing solutions*

### *C. Fuzzing process*

### *D. Fuzzing results*

## VII. Privacy

## VIII. Conclusion

## References