Permission Analysis of Health and Fitness Apps in IoT Programming Frameworks

Mehdi Nobakht*,Yulei Sui‡,Aruna Seneviratne§,Wen Hu*,
*School of Computer Science and Engineering, UNSW, Sydney, Australia
‡Faculty of Engineering and Information Technology, University of Technology Sydney, Australia
§School of Electrical Engineering and Telecommunications, UNSW, Sydney, Australia

***Abstract*—Popular IoT programming frameworks, such as Google Fit, enable third-party developers to build apps to store and retrieve user data from a variety of data sources (e.g., wearables). The problem of overprivilege stands out due to the diversity and complexity of IoT apps, and developers rushing to release apps to meet the high demand in the IoT market. Any incorrect API usage of the IoT frameworks by third-party developers can lead to security risks, especially in health and fitness apps. Protecting sensitive user information is critically important to prevent financial and psychological harms.**

This paper presents PGFIT, a static permission analysis tool that precisely and efficiently identifies overprivilege issues in third-party apps built on top of a popular IoT programming framework, Google Fit. PGFIT extracts the set of requested permission scopes and the set of used data types in Google Fit-enabled apps to determine whether the requested permission scopes are actually necessary. In this way, PGFIT serves as a quality assurance tool for developers and a privacy checker for app users. We used PGFIT to perform overprivilege analysis on a set of 20 Google Fit-enabled apps and with manual inspection, we found that 6 (30%) of them are overprivileged.

***Index Terms*—IoT Programming Frameworks; Permissions; Least Privilege; Static Program Analysis;**

I. INTRODUCTION

The Internet of Things (IoT) consists of embedded devices (e.g., wearables and smart home devices) that generate data, and end applications (e.g. mobile apps) that consume data and optionally take actions. Recently, programming frameworks have emerged which enable programmers to develop third-party apps to process such data. In particular, IoT programming frameworks for health and fitness-tracking are receiving more attention due to the popularity of wearables, smart watches and the proliferation of third-party IoT apps. Google Fit [1], Apple's HealthKit [2], Samsung Digital Health Platform [3], and Microsoft's HealthVault [4] are a few examples of such platforms.

As a representative IoT programming framework, Google Fit paves the way for third-party app programmers to efficiently write health and fitness apps by providing high-level centralized APIs, without the need to understand low-level implementation details. Google Fit also has a central cloud-based repository that allows a user to store and retrieve health and fitness-related data from multiple apps and devices (e.g., activity trackers and smart watches).

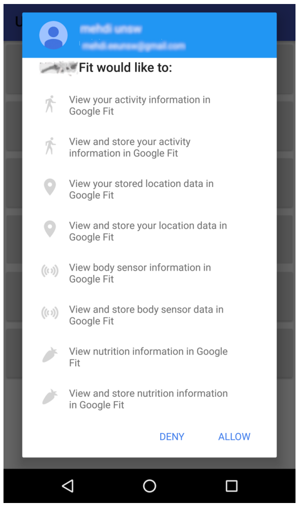
Google Fit APIs provide access to data with specified permissions associated with a Google user account. Google Fit uses OAuth protocol to authorize third party apps by obtaining consent from users to access fitness information. Authorized apps are then allowed to store or read the user's fitness information. Google Fit blends the stored data from a variety of apps and makes them available to the user and authorized apps on behalf of the user. As far as we know, apart from Google Fit's own app, 43 health and fitness apps based on the Google Fit programming framework are available on the market (e.g., exercise activities trackers, heart rate monitors and calorie counters [5]). These apps are developed by, among others, Motorola, Intel, Sony, Adidas and Nike.

An IoT app on Android needs to be authorized before it can access user health and fitness data. The Android permission system always asks the user to authorize an app by popping up a dialog screen listing the permission scopes with ”deny” and ”allow” buttons (see Figure 1). The user can choose only one or the other. Providing proper permission scopes is important to avoid overprivilege, as it exposes user to the risk of leaking private user information. Protecting sensitive user information is important, yet our studies indicate that many apps request access to more data than they actually need to perform their functions; leads the apps to significant overprivilege.

Granting the permissions in any app requires that third-party developers understand the usage of APIs in IoT frameworks. Giving unnecessary permissions to an app may expose users to privacy risks, e.g., leaking sensitive user information, such as blood pressure levels, activity data or user physical locations, to friends, health care providers or even health insurance companies. The increasing deployment of various third-party apps on top of IoT platforms raises security and privacy risks. It is important to ensure that granted permissions will not be abused. However, detecting overprivileges is challenging due to the sheer size of modern IoT apps and the complications of API usage in IoT programming frameworks.

This paper focuses on discovering overprivilege permissions to access user data within IoT platforms. In particular, we analyse a dataset of apps working with the Google Fit framework to determine how they adhere to the least-privilege principle to protect sensitive user data. To this end, we perform comprehensive studies in the Google Fit access control mechanism to gain insights into their structure and key requirements. We have summarized the scopes of permission access and data types used in existing Google Fit APIs.

We present PGFIT, a static analysis tool to identify over-



[redacted]

[redacted] Fit would like to:

View your activity information in Google Fit

View and store your activity information in Google Fit

View your stored location data in Google Fit

View and store your location data in Google Fit

View body sensor information in Google Fit

View and store body sensor data in Google Fit

View nutrition information in Google Fit

View and store nutrition information in Google Fit

Fig. 1. A Typical Google Fit Consent Screen.

privileged apps developed upon the Google Fit programming framework. We formulate the permission analysis as a source-sink graph reachability problem. PGFIT introduces context-sensitive reachability analysis into overprivilege detection in IoT apps. By considering event-driven callbacks in Android apps, PGFIT performs control-flow reachability analysis to obtain the program slices from a source node (a Google Fit API call which grants a permission scope) to a sink node (an API call which consumes data types). An overprivilege is reported in an app if a permission granted at a source node is never used (based on the data types) in any of its sink nodes. We have evaluated PGFIT on a set of 20 third party apps developed upon Google Fit. Our analysis found that 6 (30%) of them are overpriviledged.

The paper makes the following key contributions:

- We have identified the overprivilege issues in the IoT apps built upon the existing popular IoT programming framework Google Fit.
- We present PGFIT, a static analysis tool that introduces context-sensitive reachability analysis into overprivilege detection in apps on top of IoT programming frameworks.
- We have implemented PGFIT as a tool and evaluated it over a set of 20 IoT Apps built on Google Fit. With manual inspection of overprivileged apps, it turned out that 6 (30%) of them are overprivileged.

II. BACKGROUND AND OUR STUDIES OF GOOGLE FIT

Google Fit is a health and fitness-tracking platform developed by Google which uses sensors in a user's activity tracker or mobile device to record physical fitness activities such as walking or cycling. It also enables the user to measure the stored information against their fitness goals in order to provide a comprehensive view of fitness activities. Google Fit, with its open programming framework, enables third-party developers to build health and fitness apps to collect, insert, or query user fitness-related data. As such, Google Fit-enabled apps upload such data to a central repository. The stored data related to a user belongs to the user and is associated with the user's Google account. Google Fit blends a user's data collected from various sources and makes it available to the user and to authorized third-party apps from a single place at any time. In this way, users are able to query the stored fitness data and to track their progress.

A. Overview

The central repository of Google Fit can be accessed by Android and Web apps. Google Fit thus provides two sets of APIs: Android APIs and Web REST API. Google Fit APIs for Android devices have two main functionalities; ***(i)*** to provide access to data streams from sensors embedded in Android devices and sensors available in companion devices such as wearables, and ***(ii)*** to provide access to data history and to allow apps to obtain the stored data. Android apps in these two scenarios can be seen as data sources and data sinks respectively. Google Fit REST API, however, is not supposed to connect to any sensor. Thus, it is intended to access user data in the fitness store. REST API can be used by any Web browser on any platform. Since our study involves permission analysis on Android fitness apps, we focus on the Google Fit Android APIs interface.

B. Connecting to Google Play

To use Google APIs such as authentication, Map, and Google Fit, a Google-enabled app must first connect to Google Play services using Google common API. Once connected to Google Play services, the app can call Google API methods. To connect to Google APIs, a third-party app must create an instance of `GoogleAPIClient` which provides a common entry point to all Google Play. The `GoogleAPIClient` provides methods that allow the app to specify what Google APIs are required and the desired authorization scopes.

To analyse a Google Fit-enabled app, we used the publicly available Google Fit API [6] and collected all information about Google Fit API interface methods. We additionally used Google developer documents [7] and collected information regarding Google common API methods which are typically used in fitness apps. The collected specification from Google common APIs and Google Fit APIs includes method names, descriptions, and target classes. We saved the collected specification in a list to be used later to identify Google Fit API calls and Google common API calls.

More specifically, a Google-enabled app should invoke `addAPI` method and pass the required API token as an input parameter to specify what Google API is required. The required Google APIs appear as string literals in the field of `Api` class and the values of Google APIs strings are documented by Google references. For example, an app that requires access to raw sensor data streams must add *Sensors API* to enable this part of Google Play services.

C. Fitness Data Representation

Google Fit defines high-level representations for fitness data stored in its repository, in order to make it easier for apps to interact with the fitness store on any platform and to extract the required information. The Data Types representation in Google Fit abstracts away details for apps wishing to access fitness data. In this way Google Fit removes unnecessary details such as how the data is being collected or what sensors, hardware or even apps are being used.

To illustrate how this device-independent abstraction works, consider a case where a user uses two different Google Fit-enabled apps to record their activities. The first app tracks cycling activities by using sensors in a wearable device. The other app records walking activities by utilizing embedded sensors in the user's smart phone. Both apps expose raw sensor data from hardware sensors to Google Fit. Each value in such streams of data contains information about the user's activity. The user can later use a third app to extract total calories expended over a time interval. For this purpose, the third app can use `com.google.calories.expended` data type to query such information from Google Fit Store and deliver it to the user. In this way, Google Fit abstracts away any details from available data points in the fitness store.

D. Permissions and User Controls

Google Fit requires user consent before apps can access user fitness data. Apps must obtain authorization by specifying the *scope* of access to fitness data and the level of access. Google Fit classifies fitness data into four different data types: *activity*, *biometric*, *location*, and *nutrition*. The variation of fitness data types with read and write privileges creates a set of 8 different authorization scopes. For instance, the `FITNESS_ACTIVITY_READ` scope provides read-only access to all data related to a user's activities.

Google Fit provides an OAuth-based authentication service for apps to obtain required authorization scopes. OAuth service involves a multi-step authorization dialogue over HTTPS between three entities: Google Fit cloud backend, the app wishing to access fitness data, and the user who owns the fitness data. The app first must specify one or more scopes of access. Once Google Fit receives the app request, the user is prompted to grant the app the required permissions. The user must approve or deny the request at once. Figure 1 shows a consent screen for the Google Fit app developed by authors.

Once the user approves the app request to access the user's fitness data, Google Fit sends the authorization code to the app and upon app acknowledgement sends an access token. Having acquired a *scoped* OAuth bearer token, the app can make Google Fit API calls to access all fitness data types defined by that scope.

More specifically, a Google-enabled app should invoke `addScope` method with the required OAuth scope as an input parameter to specify the required authorization scope. Google Fit defines authorization scopes as string literals. In most cases, these strings are passed directly to the `addScope` method. However, in some cases, an instance of `Scope` class from Google common API is created to return Google Fit scope strings. In these cases, the constructor method of this class accepts a Uniform Resource Identifier (URI) string to indicate the intended scope. The values of URI strings are documented in Google Fit API.

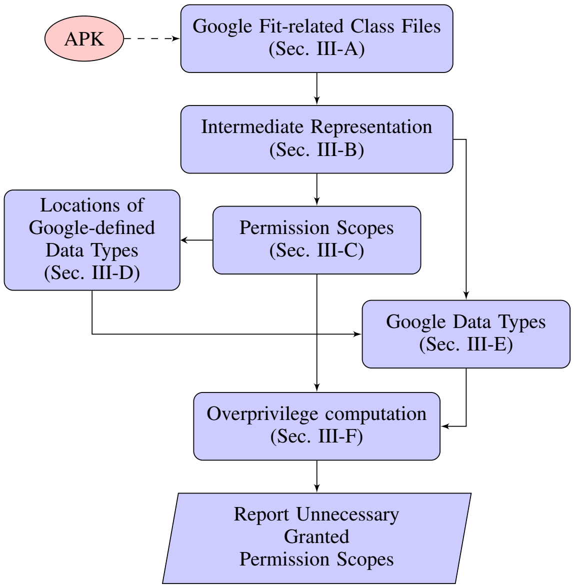

APK
Google Fit-related Class Files (Sec. III-A)
Intermediate Representation (Sec. III-B)
Locations of Google-defined Data Types (Sec. III-D)
Permission Scopes (Sec. III-C)
Google Data Types (Sec. III-E)
Overprivilege computation (Sec. III-F)
Report Unnecessary Granted Permission Scopes

Fig. 2. An overview of PGFit.

As seen in Figure 1, authorization of Google Fit-enabled apps is coarse-grained; multiple permission scopes to access fitness data types are granted at once. Thus, the user should grant permission to the app to access all requested scopes or deny all. While this coarse-grain authorization can improve the simplicity and stability of Google Fit platform, it also leaves users with no option to grant or deny permission scopes separately.

III. PGFit - Design and Implementation

We developed a static analysis tool called PGFit, which analyzes Google Fit-enabled Android apps in order to determine whether the requested authorization scopes are indeed needed. PGFit takes a given Android Application Package (APK) file as input and performs analysis on it to compute Google Fit scope overprivilege. We assume apps are not obfuscated and can be analyzed by ASM API. An overview of PGFit is shown in Figure 2 and through the following sections, each phase will be explained.

A. Identifying Google-related Class Files

In the first phase, PGFit takes a given APK as input and employs *dex2jar* [8] to generate an equivalent JAR file containing Java class files. One of the challenges in analyzing obtained JAR files from Android Google Fit-enabled apps is the sheer size of the JAR files.

To overcome this challenge, we first collected all information about Google Fit API method names and Google Fit-defined data type names from the publicly-available Google Fit APIs [6] and stored the Google Fit-related string names in a set. Then, PGFit searches for entries in the aforementioned set among all compiled class files of the JAR file. The output is a list of compiled class files related to Google Fit. In this way class files with no Google Fit API calls are filtered out.

B. Intermediate Representation

We need to convert the compiled class files to the intermediate representation suitable for our analysis. PGFit takes the list of compiled class files and visits all of their methods to extract all method invocations. This list of method invocations also includes Google API invocations which are called in the fitness app. PGFit stores this list of method invocations and other related information, including caller and callee classes and target methods along with their descriptors.

PGFit examines every method invocation in the above list to determine whether it is a Google Fit API call or belongs to Google common APIs. It takes every method invocation and compares ***(i)*** the method name, ***(ii)*** method description, and ***(iii)*** the target class of method with the Google Fit specification obtained from Google API references, as explained in Section II-B. If the invocation matches Google Fit API, PGFit, it then labels it with the corresponding Google Fit API interfaces. The list of method invocations is then refined to contain only Google common API calls or Google Fit APIs.

C. Extracting Permission Scopes

In this phase, PGFit runs multiple threads to analyze every method invocation in the list of Google Fit-related method invocations to discover ***(i)*** what Google Fit APIs have been requested to connect and ***(ii)*** what authorization scopes have been requested. PGFit obtains this information from Google common API method calls. Google Fit-enabled apps use `addAPI` and `addScope` methods to connect to the required APIs and specify the scope of access.

PGFit searches in the list of Google Fit-related method invocations for the above two methods. The method invocation which adds a scope must satisfy two conditions: ***(i)*** it must be `addScope` method of `Builder` class whose methods are used to configure the instance of `GoogleAPIClient` at the main entry point of Google Play services , and ***(ii)*** it must have either a string parameter whose value is a scope permission value or an instance of `Scope` which in turn returns Google Fit scope strings.

D. Identifying the Locations of Google-defined Data Types

In order to discover all Google-defined data types that a Google Fit-enabled app uses, we first need to identify the procedures that may consume such data types. Analyzing all procedures of the app with a large number of class files is an inefficient and resource-intensive task. Thus, PGFit aims to identify potential procedures containing Google Fit method invocations which may use Google-defined data types.

Obviously the procedure in which the app requests the required permission scopes is more likely to contain method invocations which use Google-defined data types. Thus, all the following statements and called procedures immediately after the point where the permissions are requested should be analyzed. To this end, PGFit searches for Google-defined data types in a given app by analyzing all statements and procedures after a permission scope is requested. PGFit is aware of the starting point of analysis, as explained in the previous section.

Algorithm 1: Overprivilege Computation

```
Input  : $n_{src}$
Output: Overprivilege reports for each source node
Procedure OverpriviledgeComputation($n_{src}$)
foreach $n_{src} \in SRC$ do
    $D \leftarrow$ ReachabilityAnalysis($n_{src}$) ;
    Let $D'$ be a set of Google-defined data types of $n_{src}$;
    if $D \cap D' = \emptyset$ then
        report overprivilege $n_{src}$
    end
end
```

E. Extracting Google-Defined Data Types

Once potential procedures with Google-defined data type consumption are identified, PGFit analyzes every statement of these procedures. It first builds a call graph over the procedures under examination and performs a forward traversal of the graph looking for any Google API method invocation which consumes a Google-defined data type. Arguments of a method can be passed by value or by reference. In cases where a reference to the data type is passed to the method invocation, PGFit backward traverses the graph to find the value of the reference and obtain the actual Google-defined data type.

Typically, tasks and functionalities in mobile apps are spread across several procedures and end classes. Thus, it is necessary to perform *inter-procedural analysis* [9] which operates across all class files within the app. Since the information flows both from the caller procedure to its callee and in the opposite direction, we use *call graphs* to inform which procedure calls which. A call graph is a set of nodes (vertices) and edges such that each node represents either a *call site* (a place where a procedure is invoked) or a procedure and an edge represents the connection or relationship between the call site and the procedure. More specifically, PGFit uses *Inter-procedural Control Flow Graph* (ICFG) consisting of a set of nodes and a set of edges.

A forward reachability analysisis performed to compute the data types of every source node $n_{src} \in SRC$, which is a program statement where permission scopes are requested. SNK denotes the set of program statements consuming Google-defined data types. PGFit performs a forward traversal on the ICFG of the analyzed program from every source node n_{src} to find its sinks.

F. Overprivilege Computation

Algorithm 1 computes whether a source node n_{src} is overprivileged by comparing the two sets of data types D and D', where D is computed through reachability analysis as explained in the previous section and D' is the associated data types of n_{src} immediate available from Google Fit APIs. An overprivilege is reported if $D \cap D' = \emptyset$ since an authorization permission scope in D is not permitted in D'. For example, if PGFit does not retrieve any data type (i.e., $D = \emptyset$) for a permission scope whose corresponding data types are D' , it will report an overpriviledge warning because $D \cap D' = \emptyset$.

IV. Analysis Results

We applied PGFit to a set of 20 fitness applications built upon Google Fit to identify the occurrence of overprivilege regarding authorization scopes.

A. Dataset Collection

The input to PGFit are Google Fit-enabled Android applications. We used a publicly-available tool and downloaded applications that are free and have no region restriction imposed by Google Play. In addition, since PGFit performs permission analysis by inspecting compiled class files of a fitness application, As of November 2017, there are 43 Google Fit-enabled applications available on Google Play Store. From this set of apps, we selected all the apps that are publicly available and not obfuscated as PGFit is not intended for obfuscated code. We used *dex2jar* in combination with JD tool [10] and discarded string-obfuscated applications. Overall, our dataset consisted of 20 Google Fit-enabled fitness applications.

B. Result

We have run PGFit on our dataset of 20 fitness applications. During the first phase PGFit discovers that 14 applications contained Google Fit API calls in one compiled class file, while in 6 applications Google Fit API calls and data types were distributed in more than one class. This shows that performing static analysis over all compiled class files of an application is unnecessary. For example, *Runtastic* is a health and fitness application to help users measure their activities (walking, running, jogging or biking) against goals they set. The APK file of the application contains 3,507 compiled class files, while Google Fit API methods and procedures are being employed among 3 class files.

Meanwhile, PGFit reports the fitness services that an application requested to connect to. The statistics of the extracted connection request to Google APIs are presented in Table I. As shown in the table, History API and Sessions API are at the top of the list of most prevalent in the set of 20 apps evaluated. The former enables an application to access the fitness data history that was inserted or recorded using other applications or itself. The latter provides a functionality to create sessions when a user performs a fitness activity. The Config API is another popular fitness service, employed mainly to disconnect from Google Fit. Other fitness services, which provide functionality to store fitness data, are Recording API and Sensors API. However, these fitness services are less prevalent, meaning most fitness applications read fitness data from Google Fit rather than writing the fitness data to Google Fit, which is against Google Fit principles.

We performed overprivilege analysis on 20 fitness applications in our dataset and PGFit initially reported 7 overprivileged warnings. To validate the warnings, we used JD tool to decompile the Google Fit-related compiled class files in the 7 reported overpirvileged apps. We then manually investigated the reconstructed source codes and found that one of them was a false alarm. The false alarm was caused by the conservative analysis of PGFit in identifying the location of Google-defined data types. In the false reported app, there was a procedure which can be invoked by user interaction where Google-defined data types are used.

TABLE I
Statistics of Connection request to Fitness Services in a set of 20 Google Fit-enabled Applications

Fitness Service	# of Apps
HISTORY_API	14 (70%)
SESSIONS_API	14 (70%)
CONFIG_API	12 (60%)
RECORDING_API	8 (40%)
SENSORS_API	5 (25%)

TABLE II
Unnecessary scope permissions in 20 apps

Authorization Scope	# of Apps
SCOPE_ACTIVITY_READ_WRITE	5 (83%)
SCOPE_BODY_READ_WRITE	4 (66%)
SCOPE_LOCATION_READ_WRITE	2 (33%)
SCOPE_BODY_READ	1 (16%)

Out of 20 fitness applications in our dataset, PGFit along with manual verification found 6 applications that request at least one authorization scope but never use any data types corresponding to that scope. This is undesirable, because it allows an adversary to abuse this vulnerability to leak sensitive information from a victim's fitness data. Table II reports the unnecessary permissions among these 6 applications. For example, 5 out of 20 apps grant permission scope SCOPE_ACTIVITY_READ_WRITE, but the permission is never used in these apps; that is 83% of overprivileged apps. One example of Google Fit-enabled applications that exhibits overprivilege is *8fit*; a personal trainer providing workout routines and healthy meal plans tailored to a user. This application requests access to 3 authorization scopes but does not use any data type related to any of the requested scopes.

V. related work

There are two lines of research most closely related to ours: IoT security and least-privilege principle. This section summarizes research work in IoT security and then discuss techniques which we adopted in performing security analysis.

IoT Security. In recent years, much research has been conducted in the context of IoT security. Research in this domain mainly focused on three aspects: Devices, Protocols, Platforms. In IoT device security, Ronen et al. classified attacks on IoT devices based on how the attacker deviates from the designed functionality to achieve a different effect and demonstrated potential attacks on smart lighting systems [11]. In [12], authors identified unauthorized access to the Philips Hue smart light due to plaintext data communication and proposed a network-level solution to monitor smart home network along with a machine learning detection mechanism to identify malicious activity. In IoT protocol security, researchers warned how security flaws in IoT-specific protocols such as ZigBee [13] and Zwave [14] make devices vulnerable to compromise.

Research into IoT platform security is still in the early stages. There has been effort in analyzing security concerns on programming frameworks, in particular hub-based platforms [15], [16]. More recently Fernandes et al. performed analysis of the Samsung-owned SmartThings programming framework for smart home applications [17]. Although SmartThings is a closed systems and third-party applications are run on a proprietary cloud backend, the authors managed to access the source code of applications and discovered security design flaws in SmartThings platforms and other common vulnerabilities such as revealing sensitive information caused by the lack of sufficient protection on protocols operating between the cloud backend and the client-side hub. In comparison, our work can be applied on both cloud-based and hub-based configuration and is not limited to closed source applications.

Least-privilege Principle. Limiting applications privilege can lower potential security and privacy risks, however, there is a trade-off between the complexity of the permission control model and enforcing least-privilege. This is evidenced by a large body of prior research work [18], [19]. Recently, there are a set of static analysis tools attempting to address the challenge of creating a permission map for Android such as *Stowaway* [20] and *PScout* [21]. *Stowaway* used unit testing and feedback directed API to observe the required permissions for each API call. *PScout* is another tool which improved *Stowaway* and used static reachability analysis between permission checks and API calls to extract permission specification from the Android OS source code. More recently, Backes et al. built AXPLORER [22] tool to conduct an Android permission analysis which achieves a map that is more precise.

Our work is similarly motivated, however, these previous investigations analyzed permissions granted to third-party applications to access hardware and software resources on physical devices. In contrast to prior work, we focuse on permissions granted by the user to applications to access their private data on the cloud.

VI. CONCLUSIONS

We have studied Google Fit, a popular IoT programming framework, to determine how well Google Fit-enabled third-party apps adhere to the least-privilege principle when requesting user consent to access sensitive data. Analyzing permission on these apps is challenging due to the closed-source system and the sheer size of compiled class files.

We have developed PGFIT, a static analysis tool which takes a given Google Fit-enabled app as input and extracts the set of authorization scopes along with the used Google-defined data types in an app, by performing graph reachability analysis to compute overprivilege. We applied PGFIT to 20 Android Google Fit-enabled applications to investigate how well third-party applications follow least-privilege principle and with the help of manual inspection discovered 30% of them were overprivileged. Adding support in PGFIT to identify the procedures which can be invoked via user interaction with the app or another's app interaction and may consume Google-defined data types is part of future work. This will improves PGFIT to prevent causing false alarms.

REFERENCES

[1] Google. (2017, November) Google Fit. [Online]. Available: https://www.google.com/fit/

[2] Apple. (2017, November) HealthKit. [Online]. Available: https://developer.apple.com/healthkit/

[3] Samsung. (2017, November) Samsung Digital Health. [Online]. Available: http://developer.samsung.com/health

[4] Microsoft. (2017, November) HealthVault. [Online]. Available: https://www.healthvault.com

[5] Google. (2017, November) Apps work with Google Fit. [Online]. Available: https://play.google.com/store/apps/collection/promotion_3000e6f_googlefit_all

[6] Google. (2017, November) Google Fit Developer Reference. [Online]. Available: https://developers.google.com/android/reference/com/google/android/gms/fitness/Fitness

[7] ——. (2017, November) Google Developer Reference. [Online]. Available: https://developers.google.com/android/reference/com/google/android/gms/common/package-summary

[8] (2017, November) dex2jar. [Online]. Available: https://github.com/pxb1988/dex2jar

[9] T. Reps, "Program Analysis via Graph Reachability," in *ILPS '97*, 1997, pp. 5–19.

[10] (2017, November) Java Decompiler. [Online]. Available: http://jd.benow.ca/

[11] E. Ronen and A. Shamir, "Extended Functionality Attacks on IoT Devices: The Case of Smart Lights," in *EuroS&P '16*, March 2016, pp. 3–12.

[12] M. Nobakht, V. Sivaraman, and R. Boreli, "A Host-Based Intrusion Detection and Mitigation Framework for Smart Home IoT Using OpenFlow," in *ARES*, 2016, pp. 147–156.

[13] L. Jun. (2017, November) I'm A Newbie Yet I Can Hack ZigBee - Take Unauthorized Control Over ZigBee Devices. [Online]. Available: https://www.defcon.org/html/defcon-23/dc-23-speakers.html#Li

[14] S. G. Behrang Fouladi. (2017, November) Security Evaluation of the Z-Wave Wireless Protocol. [Online]. Available: https://sensepost.com/cms/resources/conferences/2013/bh_zwave/Security%20Evaluation%20of%20Z-Wave_WP.pdf

[15] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash, "FlowFence: Practical Data Protection for Emerging IoT Application Frameworks," in *USENIX Security '16*, 2016, pp. 531–548.

[16] Y. J. Jia, Q. A. Chen, S. Wang, A. Rahmati, E. Fernandes, Z. M. Mao, and A. Prakash, "ContexIoT: Towards Providing Contextual Integrity to Appified IoT Platforms," San Diego, CA, February 2017.

[17] E. Fernandes, J. Jung, and A. Prakash, "Security Analysis of Emerging Smart Home Applications," in *S&P '16*, May 2016, pp. 636–654.

[18] A. P. Felt, S. Egelman, M. Finifter, D. Akhawe, and D. Wagner, "How to Ask for Permission," in *HotSec'12*, 2012, pp. 7–7.

[19] A. P. Felt, K. Greenwood, and D. Wagner, "The Effectiveness of Application Permissions," in *WebApps '11*, 2011, pp. 7–7.

[20] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android Permissions Demystified," in *CCS '11*, 2011, pp. 627–638.

[21] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: Analyzing the Android Permission Specification," in *CCS '12*, 2012, pp. 217–228.

[22] M. Backes, S. Bugiel, E. Derr, P. McDaniel, D. Octeau, and S. Weisgerber, "On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis," in *USENIX Security '16*, 2016, pp. 1101–1118.