Uncovering Vulnerabilities of Bluetooth Low Energy IoT from Companion Mobile Apps with Ble-Guuide

Pallavi Sivakumaran
pallavi.sivakumaran.2012@live.rhul.ac.uk
Royal Holloway, University of London
United Kingdom

Chaoshun Zuo
zuo.118@osu.edu
The Ohio State University
USA

Zhiqiang Lin
zlin@cse.ohio-state.edu
The Ohio State University
USA

Jorge Blasco
jorge.blasco.alis@upm.es
Universidad Politécnica de Madrid
Spain

ABSTRACT

Increasingly, with embedded intelligence and control, IoT devices are being adopted faster than ever. However, the IoT landscape and its security implications are not yet fully understood. This paper seeks to shed light on this by focusing on a particular type of IoT devices, namely the ones using Bluetooth Low Energy (BLE). Our contributions are two-fold: First, we present Ble-Guuide, a framework for performing mobile app-centric security issue identification. We exploit Universally Unique Identifiers (UUIDs), which underpin data transmissions in BLE, to glean rich information regarding device functionality and the underlying security issues. We combine this with information from app descriptions and BLE libraries, to identify the corresponding security vulnerabilities in BLE devices and determine the security or privacy impact they could have depending on the device functionality. Second, we present a large-scale analysis of 17,243 free, BLE-enabled Android APKs, systematically crawled from the official Google Play store. By applying Ble-Guuide to this dataset, we uncover that more than 70% of these APKs contain at least one security vulnerability. We also obtain insights into the identified security vulnerabilities and their impact.

CCS CONCEPTS

• **Security and privacy → Security protocols**; **Mobile and wireless security**; **Software reverse engineering**; **Privacy protections**; *Access control.*

KEYWORDS

IoT Security; UUID; BLE; Android.

ACM Reference Format:
Pallavi Sivakumaran, Chaoshun Zuo, Zhiqiang Lin, and Jorge Blasco. 2023. Uncovering Vulnerabilities of Bluetooth Low Energy IoT from Companion Mobile Apps with Ble-Guuide. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security (ASIA CCS '23), July 10-July 14, 2023, Melbourne, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3488932.3517410

1 INTRODUCTION

Over the past few years, billions of computing devices have been connected to the Internet to offer a wide range of functionalities including but not limited to sensing, locating, tracking, monitoring, and controlling, by way of the Internet of Things (IoT). Increasingly, this system of "smart devices" has come under scrutiny due to various security issues such as misconfiguration (e.g., use of default password in IoT cameras [10]), over-privileged apps [20], and insecure MQTT protocols [26]. However, this list is by no means complete. The IoT is comprised of a variety of platforms and communication protocols (e.g., Bluetooth, Zigbee, and WiFi), each of which have their own characteristics and security considerations. It is therefore imperative to analyze each technology individually as well as collectively, to gain a complete understanding of IoT security.

In this paper, through a mobile app-centric analysis, we systematically measure and analyze the functionality and security issues of BLE devices. We focus on Bluetooth Low Energy (BLE), a fast-growing wireless IoT technology, with more than 15 billion BLE-enabled devices shipped over the past few years [15]. Through a mobile app-centric analysis, we systematically measure and analyze the functionality and security issues of BLE devices. Doing so is by no means trivial, as an application may contain a variety of functionality, much of it external to BLE. Therefore, analyzing the APK code as a whole would result in an inaccurate view of BLE usage. Similarly, relying on vectors such as Google Play categories to derive this information would result in a coarse-grained and inaccurate view of the BLE usage too, as most apps fall under generic headings such as "Tools" or "Lifestyle", which are not indicative of the actual BLE functionality.

Key Insights and Techniques We overcome the above challenges by deriving additional knowledge from a variety of sources, including Google Play, the Bluetooth Special Interest Group (SIG), and the APK itself with a novel focal point—Universally Unique Identifiers (UUIDs), which underpin data transactions in BLE. That is, data on BLE devices are stored within data structures known as *attributes*, where each attribute is identified using a UUID. Some UUIDs have fixed meanings as defined by the Bluetooth SIG, while others are vendor/developer-defined. Because a UUID tends to represent a specific type of data, the use of UUIDs within BLE applications can provide valuable insights about the data on the device, security vulnerabilities and corresponding attacks that can be mounted against the devices or the data held on them. As such, we present a framework, Ble-Guuide, for identifying and prioritizing applications that interface with vulnerable BLE devices, as well as the first large-scale analysis of the functionality and security of BLE IoT. We do this by mining and categorizing the UUIDs from BLE-enabled apps, and performing a BLE-centric security analysis of such apps.

Ble-Guuide assigns every APK the set of functionalities based on its UUIDs, and further augments this with data obtained from the Bluetooth SIG database and Google Play using natural language processing techniques. It also performs a six-fold security analysis of the UUIDs and APKs. The combined output enables us to identify vulnerable apps that are flagged as having sensitive or critical BLE functionality.

Findings The results from our analysis by Ble-Guuide with a dataset of 12,500+ unique BLE UUIDs, extracted from 17,243 APKs, give rise to the following findings:

- **Prevalence of issues:** Our analysis showed that more than 70% of the tested applications exhibited at least one of the 6 security vulnerabilities identified by Ble-Guuide, and over 10,000 apps exhibited more than one vulnerability. We performed a series of manual analysis to verify these results and found applications that revealed the presence of unprotected user health information, had potential for PII leakage and included insecure authentication mechanisms in a security application.
- **Unauthorized data access:** Around 70% of all APKs use SIG-defined UUIDs, and 1,457 APKs use *only* SIG-defined UUIDs. These include information such as glucose data (158 apps) or blood oxygen and pressure (250 apps) that account for more than 120 million downloads. These are susceptible to unauthorized access by co-located apps in the absence of application-layer security [7, 36]. As Google states in their developer guidelines [7], and as shown by previous research [36], the data held by such UUIDs is vulnerable to unauthorized access by co-located apps in the absence of application-layer security.
- **Incorrect usage of UUIDs:** Over 50 APKs use SIG-defined UUIDs incorrectly. A number of APKs appear to use health-related services when there was no apparent reason for doing so.
- **Lack of over-the-air update:** A large percentage of applications do not contain an over-the-air firmware update mechanism. This implies that, even if serious security vulnerabilities were found in a BLE device, there would be no easy way of fixing them. In addition, more than 200 APKs, with more than 100 million downloads overall, include *insecure* firmware update processes.

2 BACKGROUND

This section describes the structure and usage of data in BLE, as well as pertinent security considerations. Note that, while several security issues have been identified with BLE over time [11, 18, 25], in this paper we focus only on the security and privacy issues of BLE data as defined by the Generic Attribute Profile (GATT) layer of BLE. We do not focus on the security of the wireless connection [11, 18, 25] nor the security of other layers that are in charge of other aspects that have security and privacy implications such as the MAC address selection [14].

2.1 Data on BLE Devices

Unlike Bluetooth Classic, BLE only handles small, discrete values of data that are known as *attributes*. The format of attributes and the underlying mechanism of how attributes are read and written are defined by the Attribute (ATT) Protocol. The Generic Attribute (GATT) Profile defines a hierarchical structuring of attributes to achieve specific purposes [15].

Attribute Structure An attribute has a specific format, consisting of a handle, a type, and a value. The handle can be thought of as the attribute's address. The type is a 128-bit UUID that describes the type of data being contained in an attribute. The value depends on the type of attribute and may hold actual data or may be used to identify the attribute [22].

Services and Characteristics GATT describes different types of attributes, the most basic being a *characteristic*. This holds a single value and is usually the type of attribute that holds the data of interest. Several related characteristics are grouped into a *service*.Services and characteristics are both types of attributes and therefore are identified using UUIDs.

Adopted vs. Custom UUIDs The Bluetooth SIG has defined some standard services and characteristics with specific meanings. This means that the associated UUIDs can be tied to the defined behaviour. We refer to this type of UUID, i.e., one that has SIG-defined functionality, as an *adopted* UUID.

As an example, the SIG has defined a Continuous Glucose Monitoring (CGM) Service for use with BLE-enabled glucose measurement devices. It describes a set of characteristics that must be implemented on a CGM device in order for the device to be able to claim conformance with the CGM Service specification. The CGM service and included characteristics have fixed UUIDs. This enables interoperability, as a connected device will know from the UUIDs the type of data that is held and the behaviour that can be expected.

All UUIDs defined by the SIG are derived from the same Base UUID [23]. A range of 2^{32} values (which use the Base UUID) is reserved by the SIG [40]. That is, UUIDs that are created by modifying the first 32 bits of the Base UUID should not be defined by vendors for their own use, although they can use the ones defined by the SIG. To obtain a custom UUID within the reserved range, vendors need to pay a fee to the SIG, which then assigns a member UUID [16]).

The Bluetooth specification allows for the creation of custom services and characteristics, where the developer has full control over the type and format of data. These services and characteristics will require *custom* UUIDs. Any 128-bit value outside the Bluetooth SIG reserved range may be used by developers to create custom UUIDs for their own services and characteristics.

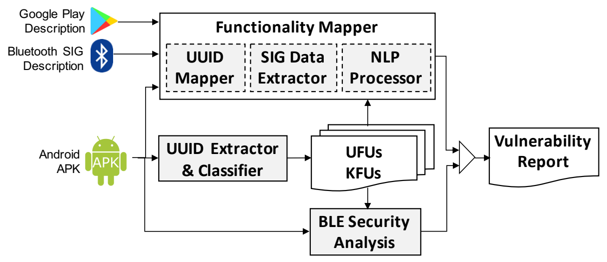

Figure 1: Overview of BLE-GUUIDE**.**

2.2 Security Issues with BLE Characteristics

We classify GATT security and privacy issues into two main groups, depending on the level of access an attacker would have to the device characteristics: reads or writes.

Characteristic Reads The BLE specification provides mechanisms to establish encrypted connections between devices via pairing. Devices that do not require pairing, or that use insecure pairing mechanisms, are susceptible to unauthorized connections from third parties [48]. Also, devices that do not use application-layer security to communicate with their corresponding app can be accessed without restrictions by co-located apps that have been granted the BLUETOOTH permission [36].

The security implications of unauthorized characteristic reads will be device specific. For instance, in a fitness tracker, this would allow attackers, to read sensitive information such as the user's heart rate or step count. For other devices this could mean access to the glucose level for a glucose meter or the device status for a smart lock.

Characteristic Writes In a similar way as for characteristic reads, a device with an insecure pairing process or lack of app-layer security can be susceptible to unauthorized writes. In this case, depending on the kind of device being affected, the impact of this security issue can be far greater. An unauthorized write on an insulin pump could result in incorrectly dispensing an insulin dose. More broadly, for devices allowing over-the-air firmware updates, this could result in modified and potentially malicious firmware images being installed.

As these examples show, the security implications of the same kind of vulnerability will be very different depending on the functionality of the affected device. Knowing the type of data held within each UUID on the device can tell us a lot about its functionality, and the corresponding security implications if such a device is vulnerable. We use this notion to develop our framework.

3 BLE-GUUIDE: A FRAMEWORK FOR PRIORITIZED SECURITY ANALYSIS OF BLE DEVICES

In this section, we describe our BLE functionality and security analysis framework, BLE-GUUIDE. The framework (depicted in Figure 1) takes as input a BLE APK, as well as data from Google Play and the Bluetooth SIG, and identifies its BLE-relevant functionality, as well as possible security vulnerabilities. It then combines both to identify, from a dataset of thousands, a smaller subset of apps where specific BLE security vulnerabilities will have more severe implications for their users. The framework comprises three main components: (*i*) **UUID Extractor & Classifier**: extracts BLE UUIDs from Android APKs and classifies them according to our custom categorization, (*ii*) **Functionality Mapper**: identifies BLE-relevant functionality within an APK, and (*iii*) **BLE Security Analysis**: performs a BLE-specific security analysis.

For both analyses, i.e., security and functionality, we use UUIDs as the starting point. Some of these UUIDs will have a publicly known functionality (e.g. UUIDs in the BLE specification) while others won't have public data available. We describe how we categorize UUIDs into *Known* and *Unknown Functionality* UUIDs (KFUs and UFUs) in section 3.1. Our technique for assigning functionality to UFUs is described in §3.2. The mapped UFU functionality, along with other inputs, is used to perform BLE-relevant functionality mapping for the overall APK, and hence the associated BLE peripheral. Finally, the security analysis performed by our system is described in §3.3.

3.1 UUID Extractor and Classifier

The functionality and security analyses performed by BLE-GUUIDE both use UUIDs as the starting point. These are first extracted from an APK before being classified according to our categorization.

UUID Extraction We adapt the algorithm described in BLEScope [48] to perform UUID extraction. Specifically, we perform permission analysis for an APK to check whether it declares BLUETOOTH permissions. If it does, then it is further analyzed to see whether it uses standard Bluetooth API calls to interact with the corresponding BLE device(s). By augmenting the algorithm described in BLEScope with getter and setter information for UUIDs, we perform UUID extraction against input APKs. The final output is stored as a JSON file, containing the UUIDs as well as the method(s) within which the UUID was found.

Categorizing BLE UUIDs We categorize UUIDs depending on whether the functionality provided by them is publicly known or not. For this, we utilize the Bluetooth SIG as our primary source of information, as the SIG describes all adopted service and characteristic UUIDs in terms of their functionality. We also use UUID information from BLE chipset manufacturers and device vendors when they are uniquely defined and publicly documented.

Known Functionality UUIDs The BLE specification provides a list of adopted services and characteristics that can be used by device manufacturers to achieve functionality such as heart rate monitoring (0x180D), insulin delivery (0x183A) or user data gathering (0x181C) among others. In addition to this, many vendors provide publicly available information about their devices and how to interact with them in the form of SDKs or other documentation. We consider the UUIDs representing these services and characteristics as *Known Functionality UUIDs* or *KFUs*. That is, they have specific assigned functionality or meaning in the context of Bluetooth Low Energy that we can derive from publicly available information.

Our KFU database includes all the adopted service and characteristics defined by the BLE specification. We also include SIG-assigned member UUIDs (which can be used as service UUIDs within their own applications), but only when unique functionality is associated with them. In addition, we include UUIDs that are uniquely defined and publicly documented by BLE chipset manufacturers or device vendors. In the case of chipset manufacturers, these UUIDs will be present, independently of the device functionality, if the device has been developed using the manufacturer toolkit. An example of these would be UUIDs defined for over-the-air (OTA) firmware updates, also referred to as device firmware update (DFU).

We obtain such KFUs from manufacturer websites or, in a few cases, from developer sites (but only if multiple sites cite the UUIDs as belonging to the same device, and no sites assign different functionality to them). We also update our list of KFUs using a "feedback" mechanism from the UFU analysis, as described below.

Since UFUs are undocumented, they will likely only be used in a single device and its associated APK. The exception to this might be in the case of device clones, where a single base device is re-branded under different names and with different apps. Taking this into account, we make the assumption that any UUID that is present in more than 5 APKs within our dataset may well be publicly documented, and perform a search for potential functionality. If such functionality is identified, we add it to the KFU list. If no functionality can be reliably discerned, then the UUID remains on the UFU list.

When performing this "feedback" analysis, we observed that some of the most common UFUs appeared to use the BLE Base UUID incorrectly. Exploring further, we found that 1,685 UUIDs used in more than 3,500 APKs (i.e., over 20% of applications in our dataset) define custom UUIDs derived from the Bluetooth Base UUID, which have not currently been assigned any functionality in Bluetooth specifications. This is despite a statement by the Bluetooth SIG that UUIDs in this range are reserved. While it may not cause issues at present, if in future these UUIDs are given a new meaning, then there is potential for conflict or confusion. For this reason, we exclude these UUIDs from our KFU list.

Unknown Functionality UUIDs We consider UUIDs that are not classified as KFUs to be *Unknown Functionality UUIDs* or *UFUs*. UFUs are typically generated by BLE device developers when they are developing BLE devices and their corresponding applications. These UUIDs are expected to be randomly generated, to avoid collisions, and there is no formal or reliable source of information that is publicly available regarding them. Note that UFUs are always custom UUIDs, while KFUs can be adopted or custom.

UUID Classification The Classifier separates UUIDs into KFUs and UFUs, where KFUs are used for deriving security implications, as described in §3.3, as well as for validating BLE-GUUIDE before it is applied to UFUs.

3.2 Functionality Mapper

In this section, we describe the functionality mapping component of BLE-GUUIDE, which identifies the BLE-relevant functionality contained within an APK, and which is thereafter used to prioritize security analyses. Our functionality mapping utilizes three primary sources of information:

(1) **Mobile Applications:** Because a UUID holds a single type of data, the functionality of a BLE device can be gleaned from the UUIDs defined within a mobile app that communicates with it. While a mobile app as a whole may contain information that is irrelevant to BLE functionality, BLE-GUUIDE employs a proximity-based method to obtain BLE-relevant information.
(2) **Bluetooth SIG:** Vendor-specific libraries used for incorporating BLE functionality into a mobile application may shed light on the type and functionality of BLE devices released by the vendor. We therefore extract libraries from mobile applications and cross-reference them against qualified products listed on the Bluetooth SIG database to provide a new vector of information.
(3) **Google Play:** Descriptions on Play provide overall information about an app's functions, including its BLE functionality, which can serve to augment the other data.

The Functionality Mapper has three sub-components: **UUID Mapper**, **SIG Data Extractor** and **NLP Processor**. Each component processes one or more of the above three data sources and produces a set of possible functional categories. The outputs of the three are then combined to form the final functionality mapping for the APK (and hence, the associated BLE device).

Building a database of functional categories In order to derive functional categories, we manually analyzed several hundred APKs, specifically their metadata, Google Play descriptions, and manufacturer websites. For each category, we provide a list of related words (e.g., microphone, speaker, camera, etc., for an `audio_visual` category). As a word can have several meanings we map each word to its corresponding WordNet definition [29]. We also include a blacklist for each word, to avoid false positives in cases where a word can be included within another word (e.g., "pulse" is a substring of "impulse"). Overall, we have 13 high-level functional categories and 40 sub-categories. The complete list of categories, with descriptions, is provided in our repository[1].

UUID Mapper BLE-GUUIDE uses Androguard [6] to extract all strings, fields and method signatures from an APK. Each of these elements is individually tested against the functional category database, to obtain element-wise lists of functional category assignments. With method signatures, we found that the combination of the class and method names produced the most accurate results. For example, for the method `Lcom/a/b/classA;->methodX(descriptor)`, we extract the `classA` and `methodX` components.

Using the entire set of functions/fields/strings found within an application would result in significant false positives. BLE-GUUIDE overcomes this by applying a proximity-based approach and considering only those method(s) that actually call the UUID. That is, it only considers the class and method names, as well as the fields/strings that are present within the method, for methods that actually *utilize* a UUID.

SIG Product Finder The Bluetooth SIG publishes details of qualified/declared components that incorporate the Bluetooth technology. If a specific product version was known, then searching for the product within the SIG Product Database would probably result in the most accurate description of BLE-specific functionality within the product (assuming such a description was provided).

[1]https://github.com/projectbtle/BLE-GUUIDE

Bluetooth products are developed by several manufacturers and in the case of common chipsets such as Nordic or Texas Instruments, the developers may be using specific libraries to access BLE devices. Because of this, the SIG product finder firsts performs a BLE library identification process. Then, it extracts identifying information from the library code (e.g. package names) and uses the SIG Product Database to map the UUIDs included in that library to a specific functionality (as defined in our functional category keywords). This allows us to identify the functionality of UFUs belonging to manufacturers with public data in the SIG Product Database.

Library Classifier: Our library identification is not based on results from tools such as *LibRadar* or *LibScout* as they do not include information about BLE libraries [12, 28]. Instead, we first compile the list of methods within which the UUIDs have been used. For all the methods, we extract the first level package name (including the second-level domains included in each package). For instance, for the call `com.example.BLEManager.getHR()` we extract `com.example`. We consider this list a first approximation to our libraries. To finetune this, We then calculate the pairwise distance between all the method calls. Our distance function compares each level of the package name and adds one per each different item at the same package name level. For instance, `com.example.BLEManager.getHR()` and `com.example.Bluetooth.HRM.getHeartRate()` would be three items apart. If we find two method signatures that are identical apart from one part of the package name, we consider them to be the same library (but re-branded).

SIG Device Descriptions: Once we obtain the BLE libraries, we perform an automated search of the SIG Product Database[2]. We first match the main package name of the library against the name of manufacturers included within the SIG Product Database (10K+). We compare the library name with companies' names using the Jaro-Winkler distance [43] which favors matches in the beginning of the string. After manual testing, we found 75% to be a good threshold to avoid false positives. To avoid mismatches with very short words, we only perform this process with library names of 3 or more characters.

We perform searches against the SIG Product Database using these terms. If a search returns more than 20 items, we consider the term to be too generic and skip it. For each item returned, we get the product name and marketing description, and input these to the NLP processor.

NLP Processor Given some descriptive text and a keyword, the NLP Processor executes the Lesk algorithm [13] to identify the meaning of the keyword within the context of the description. If the meaning matches the one included within our functional category keyword, we add the category to a list. This process is described in Algorithm 1.

Play Descriptions Google Play hosts Android applications, typically with a description of the application's primary functions. While these descriptions generally describe the overall functionality of the APK reasonably well, the fact that they encompass the application as a whole means that associating the functionality solely to the BLE component (UUID) may not be suitable. We

[2]https://launchstudio.bluetooth.com/Listings/Search

Algorithm 1: NLP based category matching

```
Result: App functional categories
description = getAppDescription(appPackage);
categories = [];
for $category \in functionalCategories$ do
    for $keyword \in category$ do
        if $keyword \in description$ then
            $m \leftarrow lesk(keyword, description)$;
            if $m \in category[keyword][meanings]$ then
                $categories^\frown\langle category\rangle$
```

use Google Play descriptions as an input to our functionality mapper to determine the functionality of an application. This is useful for cases where it is not possible to automatically determine the functionality of UUIDs within the application. For each APK, we download the Google Play description. After normalizing, translating (if non-English), tokenizing and stemming the description, we look for appearances of the keywords defined in our functional category database. Such keywords, along with the app description, are fed into the NLP Processor.

Combined Functionality We combine the information from UUIDs and the results from the NLP-processed SIG and Play descriptions to map APKs to BLE device functionalities. Our three methods work at different levels of granularity. While the *UUID Mapper* is capable of assigning functionalities directly to UUIDs, the SIG and Play outputs are at the library/application level. Because of this, we only consider UUIDs that have been assigned to a single category, but accept when the Google Play descriptions and the SIG product search return several functionalities (i.e. firmware update, heart rate measurement, etc.).

3.3 BLE Security Analysis

Our security analysis is split into multiple tasks, identifying the following security vulnerabilities: Sensitive KFUs, Anomalous Use of Adopted UUIDs, Insecure DFU,Insecure Attribute Reads and Writes, and Insecure Passkey Entry. The first three analyses are directly derived from KFUs, while the remainder involve additional processing of the APKs. For each analysis, we summarize the specific detection policy of Ble-Guuide with a summary box at the end.

Sensitive KFUs The Bluetooth SIG defines a large number of services and characteristics, covering domains from environment to health and fitness. For all SIG-defined characteristics, including health and fitness, and excluding only those concerning insulin delivery, the maximum security mandated in the specifications is protection via the standard Bluetooth pairing mechanism. If the specification is adhered to, then protection at higher layers will not be implemented for these characteristics. However, pairing/bonding alone is not sufficient protection on platforms that host multiple third-party applications such as Android [36]. This means that the BLE data within adopted UUIDs can be vulnerable to access by unauthorized apps if there is no app-layer security. That is, a malicious application could access sensitive data (e.g., heart rate or glucose measurements) from a user's BLE device, without users' awareness, including ones that deal with user with a BLE-enabled medical device. This observation is particularly concerning when the data in question is of a sensitive nature, such as heart rate, glucose measurements or an insulin delivery device configuration. To mitigate this vulnerability both the app and device should be updated to introduce app-layer security. Alternatively, developers may want to introduce application-level access control to enable other apps to access their device with the user's authorization [37].

> *A **sensitive KFU** vulnerability is detected if (i) an app uses adopted UUIDs that hold sensitive data such as user's health or PII and (ii) does not contain any app-layer security (indicated by cryptographically tainted BLE API calls).*

Anomalous Use of Adopted UUIDs Adopted UUIDs typically have clear meaning and functionality assigned to them. BLE-GUUIDE makes use of the defined functionality for adopted UUIDs to create a mapping between the UUIDs and the Google Play categories that they could be expected to fall under. Although Google Play categories don't provide fine-grained details about an app's BLE functionality, they do provide a high-level view that is sufficient to detect anomalies. For example, a Heart Rate Measurement UUID may be expected to be used in a Medical, Sports, or Health & Fitness application. However, inclusion of this UUID within a Finance application would be surprising. Adopted UUIDs are used because they enable interoperability between applications and BLE devices. An anomalous use of an adopted UUID could result in the wrong data being interpreted by another application or the device operating system. For instance, a device mistakenly using a health and fitness UUID characteristic could inadvertently interfere with health related readings, affecting trend and recommendations provided by other health related apps on the device. To prevent these scenarios developers should avoid using adopted UUIDs when implementing non-standard functionality.

BLE-GUUIDE applies the mapping to the input APK, to identify possible anomalies between the inclusion of an adopted UUID within an Android app and the functionality of the app as indicated by its Google Play category.

> *An **anomalous use of UUIDs** is identified if the adopted UUIDs in an app does not match the Google Play category as identified by* BLE-GUUIDE.

Insecure DFU Some BLE chipsets allow for Over The Air (OTA) firmware updates, i.e., updating of BLE firmware via the BLE interface itself. This process is normally referred to as a Device Firmware Update (DFU) and it enables a BLE peripheral device to have its firmware modified by receiving updated firmware from a connected BLE application.

A process for updating firmware is often necessary if bugs or security issues are discovered after a device has been released into the market. However, if the update process itself is not secure, the BLE device could be vulnerable to unauthorized firmware modifications. Different chipset vendors implement different DFU procedures, some of which have security mechanisms built-in by default, some that require configuration by developers in order to be secure, and some that have no security options. Each of these DFU procedures use and therefore can be identified by a specific set of UUIDs. Table 1 lists the DFU UUIDs by chipset vendor, with an indication as to whether the procedure has security that is built-in, developer-dependent or unavailable. We verified this manually by checking if the DFU processes used by vendors included any kind of firmware verification process.

Table 1: Firmware Update UUIDs

Manufacturer	F/W Update UUID(s)	Secur.
Nordic Legacy[5]	0000153X-1212-EFDE-1523-785FEABCD123 (X=0-4)	✗
Nordic Secure[3]	0000FE59-0000-1000-8000-00805F9B34FB	✓
	8E400001-F315-4F60-9FB8-838830DAEA50	✓
	8EC9000X-F315-4F60-9FB8-838830DAEA50 (X=1,2)	✓
	8EC90003-F315-4F60-9FB8-838830DAEA50	✓
	8EC90004-F315-4F60-9FB8-838830DAEA50	✓
Texas Instr.[2]	F000FFXX-0451-4000-B000-000000000000 (XX=C0,C1,C2,C3,C4,C5,D0,D1)	D
Qualcomm[4, 32]	00001016-D102-11E1-9B23-00025B00A5A5	D
	0000110X-D102-11E1-9B23-00025B00A5A5 (X=0,1,2)	D
Silicon Labs[35]	1D14D6EE-FD63-4FA1- BFA4-8F47B42119F0	D
	F7BF3564-FB6D-4E53-88A4-5E37E0326063	D
	984227F3-34FC-4045-A5D0-2C581F81A153	D
	4F4A2368-8CCA-451E-BFFF-CF0E2EE23E9F	D
	4CC07BCF-0868-4B32-9DAD-BA4CC41E5316	D
	25F05C0A-E917-46E9-B2A5-AA2BE1245AFE	D
Cypress[1]	0006000X-F8CE-11E4-ABF4-0002A5D5C51B (X=0,1)	D
NXP[30]	003784CF-F7E3-55B4-6C4C-9FD140100A16	✓
	013784CF-F7E3-55B4-6C4C-9FD140100A16	✓
ST BlueNRG[39]	XXXXXXX0-8506-11E3-BAA7-0800200C9A66 (XXXXXXX=8A97F7C,122E8CC,210F99F, 2691AA8,2BDC576)	D
ST STM32WB[38]	0000FE20-CC7A-482A-984A-7F2ED5B3E58F	✓
	0000FEYY-8E22-4541-9D4C-21EDAE82ED19 (YY=11,22,23,24)	✓

✗= DFU with known security issues. D = DFU with developer-dependent security. ✓= DFU with some security mechanisms by default.

> *An **insecure DFU** vulnerability is identified if (i) an APK contains UUIDs associated with insecure DFU processes and (ii) does not contain any UUID associated with DFU processes with security mechanisms.*

Insecure Attribute Reads and Writes While understanding security implications and their impact is fairly straightforward with KFUs, doing the same for UFUs requires greater effort and, in the general way, requires a case-by-case analysis. If no other information was provided, this would be a monumental task, given the potentially large UUID "space" of almost 2^{128} possible values. To focus on our analysis, we check, using information flow analysis whether the API calls that read and write information to UUIDs in that app (`readCharacteristic` and `writeCharacteristic`) are tainted with cryptographic API calls. These would mean that they implement some kind app-layer security (we do not specifically analyze the security of the implementation but check if there is any kind of app layer security present). If no taints are found, the UUIDs that access the sensitive services are identified as being vulnerable to unauthorized reads and writes. To do this, we leverage on BLE-Cryptracer [36]. First, we identify API calls related to both cryptography and attribute reads and writes (i.e. mark them as sources and sinks). Then, we use slicing to trace register values in smali code and check if cryptographic operations taint data being sent via attribute writes (e.g. encryption) or data received via attribute reads taint any cryptographic operation (e.g. decryption).

An ***insecure read*** *vulnerability is identified if the data read by* `readCharacteristic` *does not taint a cryptographic API. An* ***insecure write*** *vulnerability is detected if the data written to a device via* `writeCharacteristic` *is not tainted by a cryptographic API call.*

Insecure Passkey Entry Developers who want to protect their devices against MITM attacks and unwanted connections from other devices can make use of the pairing and bonding capabilities available in the BLE specification. In Android, bonding can be initiated directly by the app if it calls the `createBond()` method before obtaining the list of services. Otherwise, it will be initiated by the OS after receiving a `GATT INSUFFICIENT AUTHENTICATION` or `GATT INSUFFICIENT ENCRYPTION` error. In both cases, the OS will take care of the bonding unless the developer wants to modify this process by capturing the `ACTION BOND STATE CHANGED` broadcast action. One of the changes that developers can introduce during the bonding process is the PIN code that will be used to secure the BLE connection using the *Passkey Entry* association model. Developers who intercept the OS bonding process will set the PIN with a call to `setPin(byte[])`. Setting the PIN to a hardcoded value would make the connection equivalent to the *Just Works* association model, resulting in authenticated keys that are vulnerable to Man-in-the-Middle attacks.

An ***insecure Passkey Entry*** *bonding process is detected if* BLE-GUUIDE *finds a call to* `setPin(byte[])` *with a fixed constant value.*

4 RESULTS

In this section we present the experiment results of applying BLE-GUUIDE to a large dataset of APKs. For our measurement, we use a dataset of 17K+ Android apps obtained from Google Play. This dataset allows us to measure the BLE ecosystem in terms of its functionality and popularity (based on the number of downloads). We build this dataset from an initial dataset of 2 million Google Play apps crawled recently. We filter these apps using the criteria described in §3.1 to produce a dataset of 17,243 APKs. During this process, we found another set of 50K+ apps that included BLE-related permissions but only scanned to look for BLE advertisements. Although this kind of behaviour could have privacy implications (e.g. user tracking), they do not interact with BLE devices, and so were left out of this study.

4.1 Accuracy and Coverage

By executing the UUID Extractor against this dataset, we obtained 12,352 unique, valid[3] UUIDs from 16,197 APKs (i.e. valid UUIDs could not be extracted from 1,046 APKs). Ultimately, 454 KFUs and 11,898 UFUs were obtained, with 1,015 APKs having only KFUs, 7,878 APKs having only UFUs, and 7,304 APKs having both.

BLE-GUUIDE uses three different sources to categorize the BLE-relevant functionality contained within an APK: the Mobile Applications, the Bluetooth SIG database and Google Play app descriptions. Our coverage results are summarized in Table 2. Using all three information sources, we are able to find matches for 87.7% of the analyzed apps (98.3% when considering their download count).

Google Play From our initial dataset of 17K+ apps, we were able to obtain app descriptions for all except 12 of the apps (we had to use an online translator for 249 apps).

Using our NLP processor over Google Play descriptions we were able to extract functional categories for 11,734 apps, accounting for 97.1% of the overall downloads. This makes the extraction of functionality via NLP processing the method with the most coverage.

While most of the apps (25% of the total, 37.6% of the ones with a functional category identified) had a single category, in terms of downloads, most of the apps identified had three functional categories assigned (71.1% of the apps with a functional category match). This is expected, as a great number of applications had both medical and fitness categories assigned because of their access to heart-rate related data.

Table 2: Coverage of each of the functionality mapping methods and their combination. ↓= Downloads in millions.

		No Match	Match	Matched Functional Categories			
				1	2	3	4+
KFUs	Apps	9056	8187	3793	1310	370	2714
	%	52.5	47.5	22.0	7.6	2.1	15.7
	↓	1832.8	1643.9	1195.6	272.2	31.2	144.9
	%	52.7	47.3	34.4	7.8	0.9	4.2
G. Play	Apps	5509	11734	4414	3226	2516	1578
	%	31.9	68.1	25.6	18.7	14.6	9.2
	↓	102.1	3374.6	140.9	471.2	2401.5	361.0
	%	2.9	97.1	4.1	13.6	69.1	10.4
SIG	Apps	13441	3802	3316	330	132	24
	%t	78.0	22.0	19.2	1.9	0.8	0.1
	↓	3256.4	220.3	68.8	46.9	102.5	2.2
	%	93.7	6.3	2.0	1.3	2.9	0.1
UUID M.	Apps	11718	5525	3461	1877	99	88
	%	68.0	32.0	20.1	10.9	0.6	0.5
	↓	2651.9	824.9	452.4	348.8	21.3	2.4
	%	76.3	23.7	13.0	10.0	0.6	0.1
Total	Apps	2115	15128	-	-	-	-
	%	12.3	87.7	-	-	-	-
	↓	57.4	3419.3	-	-	-	-
	%	1.7	98.3	-	-	-	-

[3] A valid UUID is one that contains 32 hex digits in the form 8-4-4-4-12.

SIG Product Finder Our library classification methods identified 3,156 unique BLE libraries. Of these, we were able to obtain functional categories for 324, corresponding to a total of 220 million downloads. We manually verified those cases where we obtained more than one functional category for a library. Of these, there were fourteen false positives, originating from very common library names, which resulted in many hits in the SIG database.

Overall, The SIG Product Finder produced a very limited number of matches (22% in terms of apps but only 6.3% in terms of downloads). A manual inspection of the results showed that many of the companies that were being queried had incomplete information in the SIG product database about their products, including only the codename for the device with no further information about it. Also, the complexity of the names being used in some of the cases, made it difficult to map some of the libraries to the actual developers. As an example, we extracted *shenzhen* as one of our possible library names. However, Shenzhen is a well known location of many chip manufacturers and a search in the Bluetooth SIG database reveals 1,210 companies with *Shenzhen* on its name. Most of the functionality matches we *were* able to achieve using the SIG Product Finder were related to location and fitness. Interestingly, in terms of downloads, this method was useful in identifying almost half of the apps that interacted with environmental sensors in our dataset.

UUIDs The functionality mapping derived from KFUs is straightforward as those are defined directly by the Bluetooth SIG (except those cases we identified as anomalous in §4.4). We take advantage of this fact, and use a list of 376 KFUs as "ground truth" against which to validate our framework.

Given that three different sources of information (i.e., strings, fields and API method names) feed into the UUID Mapper, each of which will generate its own list of category-subcategory assignments, we had two main choices when selecting an overall outcome: (*i*) consider only those instances where the combined list consists of (possibly multiple instances of) a single unique category-subcategory, or (*ii*) take a majority vote over all assigned category-subcategory pairs. We found that the first option achieved coverage of 32% and an accuracy of 78%, while the second option resulted in greater coverage of 46%, but a lower accuracy of 74%. As we want to be as accurate as possible with category assignments, we opted to sacrifice coverage and chose option (*i*).

4.2 Summary

As can be seen in Figure 2, less than 30% of the apps are not affected by any security vulnerability, with more than 40% of them having at least 2 security vulnerabilities. Of those that had no security vulnerabilities, the two most predominant functional categories were authentication and location services (beacons) where 77% and 82% of apps had no security vulnerabilities identified respectively. In the case of the beacons this was mainly because of the widespread usage of a few beacon libraries that had no security vulnerabilities identified. The most predominant security vulnerabilities had to do with insecure reads and writes, which affect more than 50% of the analyzed apps (Table 3).

For each of the identified vulnerabilities, we highlight the five most affected functional categories (in terms of apps) in 3. For all security vulnerabilities, except sensitive KFUs, the distribution

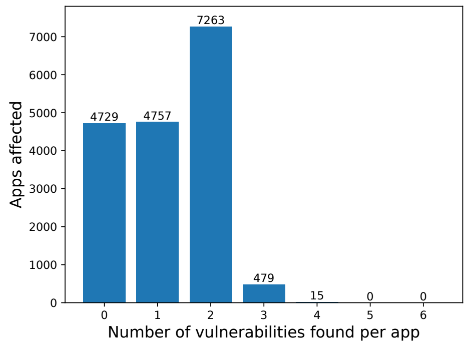

Figure 2: Distribution of apps per number of security vulnerabilities that affect them.

across the most affected functional categories is similar. For sensitive KFUs, most of the affected apps are within the fitness category, due to the high number of fitness-related UUIDs that contribute to the KFUs, as well as the amount of apps that have this functionality. Apps with security related functionality only appeared in the top 5 affected categories for two security vulnerabilities. However, these are the least prevalent vulnerabilities affecting less than 30 apps overall.

The remainder of this section describes the results obtained from the security analysis focusing on the vulnerabilities that BLE-GUUIDE is capable of detecting.

4.3 Unauthorized Access to Sensitive KFUs

The results from BLE-GUUIDE showed that 12,289 of the 16,197 APKs contain adopted UUIDs. As mentioned in §3, the data exposed via these services are vulnerable to access by malicious applications on peer BLE devices. Table 4 shows that, of the 7,077 APKs that reference BLE UUIDs (excluding GATT/GAP/common/unassigned), over 25% (2,079) are concerned with user health data such as glucose level, blood pressure and heart rate measurements. If this information is combined with other data such as activity levels, then a malicious co-located application could derive a complete health and fitness profile for a user [36]. In addition to general privacy concerns, this data could also be exploited, unbeknownst to the user, by insurance agencies and other interested parties.

4.4 Anomalous Use of Adopted UUIDs

10,556 applications within our dataset made use of at least one non-GATT/GAP adopted UUID as well as having a presence on Play at the time of testing. Only these were therefore used for anomaly detection.BLE-GUUIDE identified 333 instances of incongruous use of adopted UUIDs. From these, we separated out 123 applications that used UUIDs belonging to health-related services.

We manually analyzed 95 APKs which with no obvious need for the UUID taking the Google Play description into consideration. The purpose of our analysis was to determine whether the apps were making use of their BLE access capabilities to access sensitive data, when their functionality clearly didn't require it. Thirty one of these APKs defined UUIDs that were not used anywhere

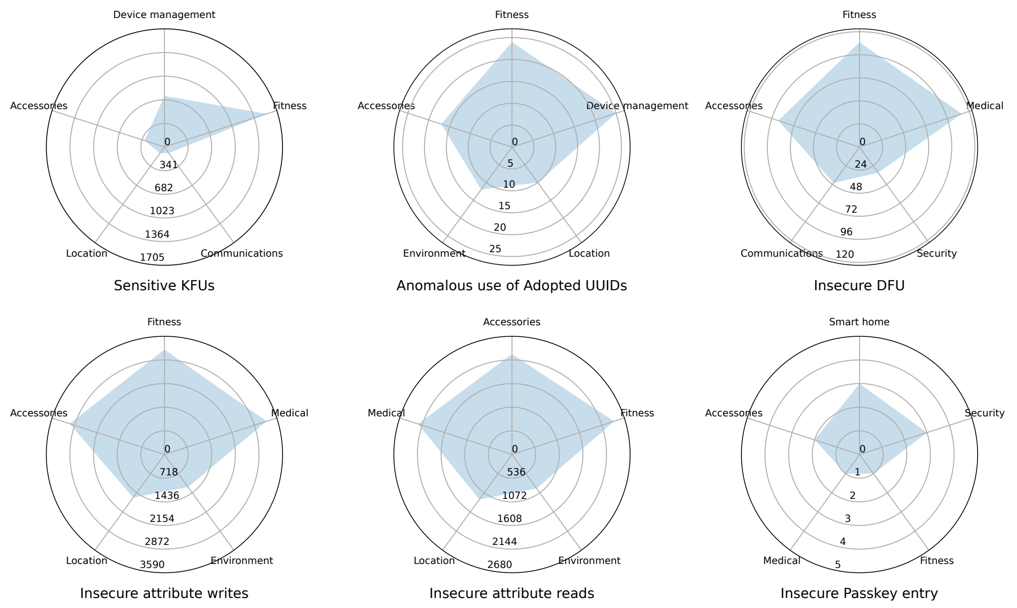

Figure 3: Five most affected functional categories (in terms of #apps) for each security vulnerability identified by Ble-Guuide.

Table 3: Number of apps (and corresponding downloads, in 1000s) detected by Ble-Guuide security analysis.

Security Issue	# Apps	# Downloads
Sensitive KFUs	1,762	76,378
Anomalous use of adopted UUIDs	123	103,920
Insecure DFU	207	115,413
Insecure attribute reads	10,267	1,657,044
Insecure attribute writes	8,593	6,923,041
Insecure Passkey Entry	12	1,010

within the app. In all such cases, the UUIDs were defined in a library, rather than in the core application. This would account for why a UUID might be present but not used, as libraries may define more functionality than what is actually used by the calling application.

There was one APK that incorporated the nRF Toolbox, which demonstrates a large number of BLE profiles. The APK itself apparently only used the DFU functionality within nRF Toolbox, but started the Heart Rate Measurement function because the code for doing so was present within nRF Toolbox.

We found one instance of a Blood Pressure Sensor service being defined within the app code. A closer analysis of this APK revealed that the code of this service was being reused (including the name) from another app of the same developer that actually connected to a blood pressure monitor.

There were a number of instances of incorrect UUID usage, i.e., UUIDs used for purposes other than for what they were defined. In particular, 4 APKs, all from the same developer, denote the Heart Rate Service as “RX Service”; another APK uses it to store door lock parameters; and one APK uses various adopted UUIDs to control a barometer.

There was also an unusual combination of UUIDs being used in a set of 42 applications which all enabled the control of music streaming to speakers. The applications used the service UUID 0x180A (Device Information) in conjunction with the Heart Rate Measurement characteristic. Given that this combination of service and characteristic does not conform to the BLE service specification, it is unlikely that this can be used for malicious data gathering. Therefore, we assume this to be a developer error.

One APK included functionality to interface with a popular fitness tracker, but with no indication of doing so within the application description. We believe that, whenever a developer intends for their application to interface with any sort of user device, particularly one that handles sensitive user data, some mention should be made within the app description.

Finally, there were ~10 APKs where health-related services appeared to be defined within the application’s own code, for no apparent reason and where the developer did not have other health-related apps on Play. It should be noted that around half of them define the Heart Rate Service, which is the BLE service most commonly used in coding examples. Therefore, it is possible to assume that the UUIDs for the service were copied from online sources unintentionally [21]. We emulated a BLE peripheral with the services declared in each APK, to check if the application would attempt to surreptitiously connect to it, but found that none did. This finding supports our assumption of accidental use of adopted UUIDs.

4.5 Insecure DFU

Out of the 16,197 APKs in our dataset that returned at least one UUID during the extraction phase, 603 APKs contained references to DFU UUIDs. Ble-Guuide identified that Nordic DFU UUIDs are the most prevalent one. Further analysis showed that 207 APKs contained only the Legacy (i.e., insecure) DFU UUIDs.This means that the devices associated with these 207 APKs, which correspond to over 100 million total downloads and potentially as many BLE devices, are vulnerable to unauthorized firmware upgrades. Forty of the 207 APKs also contained at least one health or fitness-related adopted UUID. Malicious modifications of firmware on such devices could enable sustained incorrect health information being fed to the user, with potentially life-threatening outcomes.

Note that in many cases, a peripheral has to be put into DFU mode for it to advertise the DFU UUIDs and accept a firmware image, which could be viewed as a deterrent for the attacker. However, we have observed instances where this mode is enabled through writing some particular byte sequence to another characteristic on the device. A sufficiently-determined attacker would be able to obtain this mechanism via reverse engineering or eavesdropping on the BLE communications.

4.6 Insecure Attribute Reads and Writes

We identified 8,593 APKs with no protection for BLE reads and 10,420 APKs with no protections for BLE writes. These were prioritized using the results from the Ble-Guuide Functionality Mapping component according to the perceived sensitivity of the BLE functionality. We place particular focus on functionality that relates to user health or personal data, as well as any that have security consequences, such as firmware updates. We next present a selection of case studies based on our analysis of such applications.

Case Study - ECG Applications For UUIDs that were classified as medical:measurement, we executed the open-source tool BLE-Cryptracer [36] against their host applications to identify any that did not implement end-to-end protection for medical data. We prioritized the resultant set of applications based on their installation counts and manually analyzed the most downloaded apps.

Two ECG measurement applications within the dataset read data from external ECG recording devices and display the results within the app, but with no protection between the two endpoints. According to previous research, this could mean that any other app on the same Android device with BLUETOOTH permissions can access this information [36]. We informed the developers of both applications but have received no response.

Case Study - User PII The functionality mapping phase of Ble-Guuide returned two UUIDs that were mapped to the sub-category PII (Personally Identifiable Information). Both belonged to the same application: a proximity-based “friend-finder”. Manually analyzing the app, we found code suggesting that the app advertises the user’s first name, device hardware address, and an ID within BLE advertisements, and that it also scans for these advertisements to identify other app users in the vicinity. The app also maintains a *Last Seen* parameter for each user it identifies, which can facilitate unauthorized tracking of users.

When we installed the app, we found the functionality to be rudimentary, with no user name being collected and no BLE advertisements being issued. We observed that it was in demo mode, which resulted in the data-collection functionality not being executed. However, we believe that the presence of code within the application for advertising user PII, along with burgeoning interest in user/device trackers [42], signals an increase in the future of such tracking or “finding” apps.

Case Study - Smart Door Lock From the results of Ble-Guuide, we identified an application that interfaced with a BLE-enabled door lock in an insecure manner. Specifically, the application code logs the start of an authentication sequence, and the data that is read from the BLE lock is sent to a decryption method. However, the decryption code revealed that it did not employ any standardized algorithm, but rather a custom scheme involving hard-coded arrays of decimal digits (which represent an ASCII string) and a fixed 5-character string, which represented some form of key. We notified the developersbut have received no response.

In general, the use of non-standard cryptographic algorithms (including fixed keys) is discouraged as their security has likely not been verified by a community of experts.

4.7 Insecure Passkey Entry

Twelve APKs were identified with calls to the `setPin` method with a fixed byte array input. Manual analysis showed that five such instances were for Bluetooth Classic communications, which uses the same API call for pairing. One of these applications controls the functionality of a smart skateboard, which has serious safety implications in the event of a Man-in-the-Middle attack.

Of the APKs where `setPin` was called with a BLE device, one interfaced with a Blood Pressure monitor and set the PIN to all zeros, effectively equating the *Passkey Entry* pairing model to the less secure *Just Works* model. This APK is no longer available on Google Play. We also found an app that acted as an IoT hub for controlling a variety of devices and which uses a fixed PIN when pairing with an activity wristband. Another APK was specifically designed to update the firmware on a smart lock but used the fixed PIN in the absence of a PIN setting on the lock, restricting the MitM attack window to the first time the lock is used. The remainder of the apps were used to access non-sensitive data from LED lamps, a driving aid and a music amplifier. Note that the absence of calls to `setPin` does not provide an indication of the security (or even the presence) of pairing for the remaining applications.

5 LIMITATIONS

UUIDs are sometimes generated over multiple iterations, which makes it difficult to extract them without complex static or dynamic analyses. This means that there is a possibility that we may not have obtained complete coverage of all UUIDs used by an APK. In addition, as discussed in §4.4, sometimes a large number of UUIDs may be defined within an APK, but only a small subset may get used. The extraction mechanism may not always capture this scenario, and may therefore extract even those UUIDs that are not used. In addition, if a single method declares and calls all UUIDs, then they would all be assigned exactly the same categories even if they differed in functionality.

In the case of functionality mapping, the richness and accuracy of information obtained through the various sources (i.e., API, strings, Play and SIG descriptions) depends entirely on the BLE device/app developer actually publishing such information and not obfuscating their app.

Table 4: Prevalence of Relevant Adopted BLE Services. ↓=Downloads in thousands.

Service	# Apps	↓	Service	# Apps	↓
Battery	1749	319015	**Heart Rate M.**	**1672**	**173644**
GAP	898	127936	**Health Thermometer**	**100**	**2682**
GATT	292	23979	**Blood Pressure**	**181**	**115825**
Glucose	**158**	**5787**	**Continuous Glucose M.**	**23**	**30**
Body Composition	**113**	**1561**	Running Speed	110	5155
Pulse Oximeter	**69**	**720**	**Weight Scale**	**90**	**103237**
User Data	**63**	**811**	**Insulin Delivery**	**1**	**1**

6 RELATED WORK

Bluetooth Low Energy has gained popularity in the IoT arena, and recently so too has its security. Earlier research focused on the security of the pairing process [33, 34], followed by privacy studies [19] and research into the possibility of Man-in-the-Middle attacks [25], both are which are related in different ways to BLE advertisements. In 2020, Zhang et al. [45] identified a series of vulnerabilities that would allow attackers to downgrade secure BLE connections to vulnerable modes such as ”Just Works”. The vulnerability originated from how the Secure Connections Only (SCO) mode is implemented by most operating systems (OS). Because most pairing and bonding operations are handled by the OS without the intervention of the app, a peripheral BLE device connecting to a vulnerable OS could have their connection downgraded without being aware of it. As BLE-GUUIDE focuses on security issues that originate on the BLE peripheral, a peripheral with no security issues according to BLE-GUUIDE, could still be susceptible to these attacks if connected to a vulnerable central device. Some privacy studies have also incorporated UUID analysis, e.g., Zuo et al. [48] and Celosia et al. [17] explored the possibility of fingerprinting BLE devices by using their UUIDs. These works focus on how KFUs can be used to track BLE devices and their owners. Our work also uses UUID analysis as a central element, but we expand the analysis performed in [17, 48] by addressing a wider range of security vulnerabilities in peripherals and using the UUIDs to establish the functionality of a peripheral to prioritize its security analysis. Most recently, Zhang et al. [44] showed a protocol level flaw of BLE MAC address generation, allowing attackers to deanonymize MAC addresses based on an allowlist side channel.

A number of efforts have focused on analyzing mobile apps within the BLE eco-system, from which various security vulnerabilities have been found. Sivakumaran and Blasco [36] studied the feasibility of BLE data access by unauthorized apps, while Korolova, et al. [27] described the possibility of device tracking across apps. Wen et al. [41] analyzed the over 700 Bluetooth firmware and uncovered a variety of link-layer vulnerabilities. Zhao et al. [46] analysed the security of 1,160 Android apps that act as a BLE-peripheral (instead of a central device). They found that 69% regularly broadcast device or personal information in plaintext.

There is also a large body of research for applying natural language processing (NLP) and machine learning (ML) to mobile application security analyses. WHYPER [31] made a first step towards this direction, and it checked the consistency between app permission and app description using NLP. Zimmeck et al. [47] used a keyword-based approach with a ML classifier to infer privacy policies. PolicyLint [8] analyzes policy contradictions by identifying both positive and negative statements in policy descriptions using NLP, from which to uncover misleading policy descriptions. POLICHECK [9] detects flow-to-policy consistency between app implementation and policy description to determine whether a mobile app properly discloses its privacy-sensitive behavior. Recently, Hu et al. [24] used natural language processing to analyze app store reviews. Their results show that negative app store reviews can be used to identify apps that violate app market policies.

7 CONCLUSION

We have presented BLE-GUUIDE, a framework which employs a mobile app-centric approach to identify security vulnerabilities in BLE peripherals through the UUIDs used with BLE data. The framework uses a wide variety of data sources, including NLP-processed Google Play descriptions and data from the Bluetooth Special Interest Group, as well as the application code, to determine the functionality of, and security vulnerabilities that affect, BLE devices that interface with a mobile app. The combination of functionality and security analyses allows for quick identification of devices where security vulnerabilities can have a severe impact.

Our framework identified numerous vulnerabilities, including the widespread use of UUIDs that run the risk of unauthorized data access, insecure firmware updates and potential functionality conflicts. Combined, these security vulnerabilities affect more than 60% of the tested apps, including those with sensitive functionalities. We also present a number of case studies based on manual analysis of applications that were flagged by our framework as being vulnerable. Our analyses reveal potential leakage of user health information and PII, as well as severe vulnerabilities within a smart door lock application, which has obvious ramifications for user safety. This shows that basic security problems still exist within IoT despite the increase in public scrutiny.

ACKNOWLEDGMENTS

This research was partially supported by the Engineering and Physical Sciences Research Council (EPSRC) and the UK government as part of the Centre for Doctoral Training in Cyber Security at Royal Holloway, University of London (EP/P009301/1), and by the NSF Awards 2112471 and 2226443.

REFERENCES

[1] 2015. BLE External Memory Bootloader and Bootloadable. [Online]. Available: http://www.cypress.com/file/228556/download. [Accessed: 21 May 2022].

[2] 2016. OAD Profile. [Online]. Available: https://www.ti.com/tool/download/SIMPLELINK-CC2640R2-SDK/1.40.00.45. [Accessed: 01 May 2022].

[3] 2017. Buttonless Secure DFU Service. [Online]. Available: https://infocenter.nordicsemi.com/index.jsp?topic=%2Fcom.nordic.infocenter.sdk5.v14.0.0%2Fservice_dfu.html. [Accessed: 20 May 2022].

[4] 2017. Consider blocklisting Qualcomm CSR firmware update service. [Online]. Available: https://github.com/WebBluetoothCG/registries/issues/20. [Accessed: 25 May 2022].

[5] 2017. Device Firmware Update Service. [Online]. Available: https://infocenter.nordicsemi.com/index.jsp?topic=%2Fcom.nordic.infocenter.sdk5.v11.0.0%2Fgroup__ble__sdk__srv__dfu.html. [Accessed: 25 May 2022].

[6] 2018. Androguard. https://github.com/androguard/androguard.

[7] 2020. Bluetooth low energy overview. [Online]. Available: https://developer.android.com/guide/topics/connectivity/bluetooth-le. [Accessed: 01 Feb 2022].

[8] Benjamin Andow, Samin Yaseer Mahmud, Wenyu Wang, Justin Whitaker, William Enck, Bradley Reaves, Kapil Singh, and Tao Xie. 2019. PolicyLint: Investigating Internal Privacy Policy Contradictions on Google Play. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 585–602. https://www.usenix.org/conference/usenixsecurity19/presentation/andow

[9] Benjamin Andow, Samin Yaseer Mahmud, Justin Whitaker, William Enck, Bradley Reaves, Kapil Singh, and Serge Egelman. 2020. Actions Speak Louder than Words: Entity-Sensitive Privacy Policy and Data Flow Analysis with POLICHECK. In *29th {USENIX} Security Symposium ({USENIX} Security 2020)*.

[10] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. 2017. Understanding the Mirai Botnet. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 1093–1110. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis

[11] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper B Rasmussen. 2019. The KNOB is Broken: Exploiting Low Entropy in the Encryption Key Negotiation Of Bluetooth BR/EDR. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, 1047–1061.

[12] Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 356–367.

[13] Satanjeev Banerjee and Ted Pedersen. 2002. An adapted Lesk algorithm for word sense disambiguation using WordNet. In *International conference on intelligent text processing and computational linguistics*. Springer, 136–145.

[14] Johannes K Becker, David Li, and David Starobinski. 2019. Tracking anonymized bluetooth devices. *Proceedings on Privacy Enhancing Technologies* 2019, 3 (2019), 50–65.

[15] Bluetooth Special Interest Group. 2018. Bluetooth Market Update 2018.

[16] Bluetooth Special Interest Group. 2020. 16 Bit UUIDs for Members. [Online]. Available: https://www.bluetooth.com/specifications/assigned-numbers/16-bit-uuids-for-members/ [Accessed 28 May 2020].

[17] Guillaume Celosia and Mathieu Cunche. 2019. Fingerprinting Bluetooth-Low-Energy Devices Based on the Generic Attribute Profile. In *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things*. ACM, 24–31.

[18] Timothy Claeys, Franck Rousseau, Boris Simunovic, and Bernard Tourancheau. 2019. Thermal covert channel in Bluetooth Low Energy networks. In *WiSec*. 267–276.

[19] Kassem Fawaz, Kyu-Han Kim, and Kang G Shin. 2016. Protecting Privacy of BLE Device Users. In *USENIX Security Symposium*. 1205–1221.

[20] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. 2016. Security analysis of emerging smart home applications. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 636–654.

[21] Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. 2017. Stack overflow considered harmful? the impact of copy&paste on android application security. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 121–136.

[22] Robin Heydon. 2013. *Bluetooth Low Energy: The Developer's Handbook*. Upper Saddle River, N.J. : Prentice Hall.

[23] Robin Heydon. 2016. An Introduction to Bluetooth Low Energy. [Online]. Available: https://datatracker.ietf.org/meeting/interim-2016-t2trg-02/materials/slides-interim-2016-t2trg-2-7. [Accessed: 18 May 2022].

[24] Yangyu Hu, Haoyu Wang, Tiantong Ji, Xusheng Xiao, Xiapu Luo, Peng Gao, and Yao Guo. 2021. CHAMP: Characterizing Undesired App Behaviors from User Comments Based on Market Policies. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 933–945. https://doi.org/10.1109/ICSE43902.2021.00089

[25] Sławomir Jasek. 2016. Gattacking Bluetooth smart devices. In *Black Hat USA Conference*.

[26] Yan Jia, Luyi Xing, Yuhang Mao, Dongfang Zhao, XiaoFeng Wang, Shangru Zhao, and Yuqing Zhang. 2020. Burglars' IoT Paradise: Understanding and Mitigating Security Risks of General Messaging Protocols on IoT Clouds. In *2020 IEEE Symposium on Security and Privacy (SP)*. 838–854.

[27] Aleksandra Korolova and Vinod Sharma. 2017. Cross-App Tracking via Nearby Bluetooth Low Energy Devices. In *PrivacyCon 2017*. Federal Trade Commission.

[28] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. 2016. LibRadar: fast and accurate detection of third-party libraries in Android apps. In *Proceedings of the 38th international conference on software engineering companion*. 653–656.

[29] George A Miller. 1995. WordNet: a lexical database for English. *Commun. ACM* 38, 11 (1995), 39–41.

[30] NXP. 2018. QN902x OTA Profile Guide. [Online]. Available: https://www.nxp.com/docs/en/user-guide/UM10993.pdf [Accessed 07 Feb 2020].

[31] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. 2013. {WHYPER}: Towards Automating Risk Assessment of Mobile Applications. In *Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13)*. 527–542.

[32] Qualcomm Technologies. 2016. OTAU CSR102x. [Online]. Available: https://developer.qualcomm.com/qfile/34081/csr102x_otau_overview.pdf [Accessed 09 Aug 2019].

[33] Tomas Rosa. 2013. Bypassing Passkey Authentication in Bluetooth Low Energy. *IACR Cryptology ePrint Archive* 2013 (2013), 309.

[34] Mike Ryan. 2013. Bluetooth: With Low Energy Comes Low Security. In *7th USENIX Workshop on Offensive Technologies, WOOT '13, Washington, D.C., USA, August 13, 2013*. https://www.usenix.org/conference/woot13/workshop-program/presentation/ryan

[35] Silicon Labs. 2018. AN1086: Using the Gecko Bootloader with the Silicon Labs Bluetooth Applications. [Online]. Available: https://www.silabs.com/documents/public/application-notes/an1086-gecko-bootloader-bluetooth.pdf [Accessed 04 May 2022].

[36] Pallavi Sivakumaran and Jorge Blasco. 2019. A Study of the Feasibility of Co-located App Attacks against BLE and a Large-Scale Analysis of the Current Application-Layer Security Landscape. In *28th USENIX Security Symposium USENIX Security 19)*.

[37] Pallavi Sivakumaran and Jorge Blasco Alis. 2021. Whos Accessing My Data? Application-Level Access Control for Bluetooth Low Energy. In *Proceedings of the 17th EAI International Conference on Security and Privacy in Communication Networks*. Springer.

[38] ST Microelectronics. 2016. AN4869 Application Note. [Online]. Available: https://www.st.com/resource/en/application_note/dm00293821.pdf [Accessed 01 May 2022].

[39] ST Microelectronics. 2019. BlueSTSDK. [Online]. Available: https://github.com/STMicroelectronics/BlueSTSDK_GUI_iOS [Accessed 03 May 2022].

[40] Kevin Townsend, Carles Cufí, Robert Davidson, et al. 2014. *Getting started with Bluetooth Low Energy: tools and techniques for low-power networking*. O'Reilly Media, Inc.

[41] Haohuang Wen, Zhiqiang Lin, and Yinqian Zhang. 2020. FirmXRay: Detecting Bluetooth Link Layer Vulnerabilities from Bare-Metal Firmware. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*.

[42] Lance Whitney. 2019. How to Locate Your Friends With the Apple 'Find My' App. [Online]. Available: https://uk.pcmag.com/gallery/123522/how-to-locate-your-friends-with-the-apple-find-my-app. [Accessed: 03 Mar 2022].

[43] William E Winkler. 1990. String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage. (1990).

[44] Yue Zhang and Zhiqiang Lin. 2022. When Good Becomes Evil: Tracking Bluetooth Low Energy Devices via Allowlist-based Side Channel and Its Countermeasure. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 3181–3194.

[45] Yue Zhang, Jian Weng, Rajib Dey, Yier Jin, Zhiqiang Lin, and Xinwen Fu. 2020. Breaking Secure Pairing of Bluetooth Low Energy Using Downgrade Attacks. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 37–54. https://www.usenix.org/conference/usenixsecurity20/presentation/zhang-yue

[46] Qingchuan Zhao, Chaoshun Zuo, Jorge Blasco, and Zhiqiang Lin. 2022. PeriScope: Comprehensive Vulnerability Analysis of Mobile App-Defined Bluetooth Peripherals. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security* (Nagasaki, Japan) *(ASIA CCS '22)*. Association for Computing Machinery, New York, NY, USA, 521–533. https://doi.org/10.1145/3488932.3517410

[47] Sebastian Zimmeck, Ziqi Wang, Lieyong Zou, Roger Iyengar, Bin Liu, Florian Schaub, Shomir Wilson, Norman Sadeh, Steven Bellovin, and Joel Reidenberg. 2016. Automated analysis of privacy requirements for mobile apps. In *2016 AAAI Fall Symposium Series*.

[48] Chaoshun Zuo, Haohuang Wen, Zhiqiang Lin, and Yinqian Zhang. 2019. Automatic Fingerprinting of Vulnerable BLE IoT Devices with Static UUIDs from Mobile Apps. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM.