Identifying the BLE Misconfigurations of IoT Devices through Companion Mobile Apps

Jianqi Du*†, Fenghao Xu‡(✉), Chennan Zhang*†, Zidong Zhang*†, Xiaoyin Liu*†, Pengcheng Ren*†,
Wenrui Diao*†(✉), Shanqing Guo*†, and Kehuan Zhang‡
*School of Cyber Science and Technology, Shandong University
{dujianqi, zcn, kee1ongz, liuxiaoyin, 201911899}@mail.sdu.edu.cn, {diaowenrui, guoshanqing}@sdu.edu.cn
†Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University
‡The Chinese University of Hong Kong, xf016@link.cuhk.edu.hk, khzhang@ie.cuhk.edu.hk

2022 19th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON) | 978-1-6654-8643-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/SECON55815.2022.9918597

***Abstract*—Bluetooth Low Energy (BLE) is widely deployed and has become the de-facto communication standard in the IoT ecosystem. Naturally, the security of BLE received much attention from both researchers and attackers. In another aspect, the BLE specifications provide the security guidelines for BLE deployments. Due to various reasons, the developers do not follow the guidelines in the implementation process, which introduces the misconfiguration issue. However, identifying these BLE misconfigurations in IoT device firmware is quite challenging. In this work, we do not handle the BLE-enabled devices directly. Instead, we focus on the security misconfiguration issues in their companion mobile apps, which can reflect the deployment conditions of the corresponding devices. Further, we designed an analysis tool – BSC-Checker to detect the misconfigurations based on pre-defined checking strategies. With BSC-Checker, we conducted large-scale experiments on 4,589 apps from multiple app markets. The result shows that the BLE configurations of most BLE apps disobey at least one security rule, and the current BLE deployment status is not optimistic.**

I. Introduction

Bluetooth Low Energy (BLE) is a wireless network technology designed for low-power and low-cost devices. Currently, it has become the de-facto communication standard in the IoT ecosystem. According to a recent statistics report, there are more than 8 billion BLE-enabled devices all over the world [7], supporting the deployment of smart home, health management, media sharing, sensing network, etc.

Naturally, the popularity of BLE has received much attention in the security research community. Its protocol design has been carefully reviewed from various aspects, and multiple flaws were identified, preventing the potential security risks of pairing downgrade attack [29], privilege escalation [28], and data spoofing [27]. These previous works will be reviewed in detail in Section VII. In addition, the BLE specifications [8] provide the deployment guidelines for data encryption, authentication, and authorization in the communication. However, due to the tight product cycle and the lack of security knowledge, BLE product vendors and developers may not implement the required security practices correctly, introducing the *security mis-configurations*. For example, every BLE device will be assigned a Bluetooth MAC address for data communication. If such an address keeps static, the device user will face the privacy risk of identity tracking. A more secure practice is to deploy the dynamic address generation technique. However, on the whole, there is still a lack of rounded analysis on the security misconfiguration issues of the BLE deployments.

Motivation. In the IoT ecosystem, the BLE-enabled device is often delivered with a companion mobile app to facilitate user control, called *BLE app* for short. Since this mobile app needs to communicate with the IoT device through BLE, it has (almost) the same BLE configurations as the device. On another aspect, binary-based device firmware analysis is quite challenging. Even worse, the corresponding firmware is usually not available. Therefore, from the feasibility analysis, the BLE apps can be treated as the mirror of the BLE configurations in BLE-enabled devices.

Our Work. Inspired by the typical <device-app> communication architecture, in this work, we do not handle the challenging firmware analysis. Instead, we focus on analyzing the BLE configurations on the companion mobile apps of BLE-enabled IoT devices. As mentioned above, the BLE configurations of BLE apps can reflect the conditions of BLE-enabled devices indirectly. This work aims to investigate the wild status of security misconfiguration issues of the BLE deployments in the IoT ecosystem.

Concretely, we designed an analysis framework – BSC-Checker, which takes an APK file as the input and outputs a misconfiguration report. BSC-Checker contains three main modules: 1) data collection and pre-processing; 2) security configuration checking; 3) report generation. Note that we built a series of configuration detection rules based on the suggestions of the official documents and previous research, covering the configuration requirements of Bluetooth MAC address, UUID (Universal Unique Identifier), pairing method, data encryption, and device authentication. Under these misconfigurations, it will bring various security risks, like identity tracking, eavesdropping, and unauthorized accessing.

With BSC-Checker, we conducted a large-scale experiment on 226,181 apps collected from multiple app markets. After filtering, 4,589 BLE apps were used for further security analysis. The results show that most BLE apps still use static Bluetooth MAC addresses (4,002, 87.2%) and static UUIDs (3,954, 86.2%). The pairing methods of 93.2% apps are based on the insecure “Just Work” mode or even do no deploy pairing checking. On the aspect of data transmission, the data

Authorized licensed use limited to: SHANDONG UNIVERSITY. Downloaded on November 02,2022 at 14:14:45 UTC from IEEE Xplore. Restrictions apply.

encryption is rarely deployed, saying less than 14%. Worse yet, rare apps implement application layer authentication. To sum up, the BLE configurations of most BLE apps disobey at least one security rule. Accordingly, the configuration situation of the BLE deployments in the wild is not optimistic. To demonstrate the immediate security risks and the effectiveness of our framework, we provide two concrete case studies in Section V.

Contributions. The main contributions of this paper are:

- *Problem Identification.* This work studied the security misconfiguration problems of the BLE deployments and summarized five detection strategies.
- *Tool Design.* We designed an analysis tool, BSC-Checker, for detecting the security misconfigurations in BLE apps, which can reflect the corresponding BLE-enabled IoT device conditions.
- *Evaluation and Measurement.* With BSC-Checker, we conducted a large-scale experiment on 4,589 BLE apps. The result shows that misconfiguration problems are quite common in the BLE deployments.

Roadmap. The rest of this paper is organized as follows. Section II provides the necessary background of BLE. In Section III, based on the threat model and identified security risks, we overview the security misconfigurations under detection. Section IV gives the detailed design of BSC-Checker. The results are summarized in Section V. Some limitations of this work are discussed in Section VI. Section VII reviews the related work, and Section VIII concludes this paper.

II. Bluetooth Low Energy

This section presents the necessary background about Bluetooth Low Energy (BLE), focusing on BLE workflow and Bluetooth MAC address.

BLE was first introduced in Bluetooth Specification 4.0 by the Bluetooth Special Interest Group (Bluetooth SIG) [5]. In contrast to Bluetooth Classic, BLE is designed for significantly lower power consumption. It has been widely deployed in energy-constrained IoT devices with many practical applications, such as healthcare, smart home, fitness, and beacons.

A. BLE Workflow

In the design of BLE, the typical communication between two devices involves three stages: (1) *Connection*, (2) *Pairing and Bonding*, and (3) *Communication* [8]. Also, during this process, one device acts as BLE *master* (e.g., mobile phone), and the other one as *slave* (e.g., IoT device). Figure 1 illustrates such a workflow. Note that the behaviors of the phone are controlled by the BLE apps running on the phone through Bluetooth APIs [2]. Therefore, three parties are involved in this figure.

1) Connection Stage. When a smartphone wishes to establish a connection with an IoT device, the IoT device must be in the discovery mode. The device under this mode will broadcast advertising packets to nearby devices to indicate its availability. These advertising packets contain identity information such as the device name, Bluetooth MAC address, and primary services.

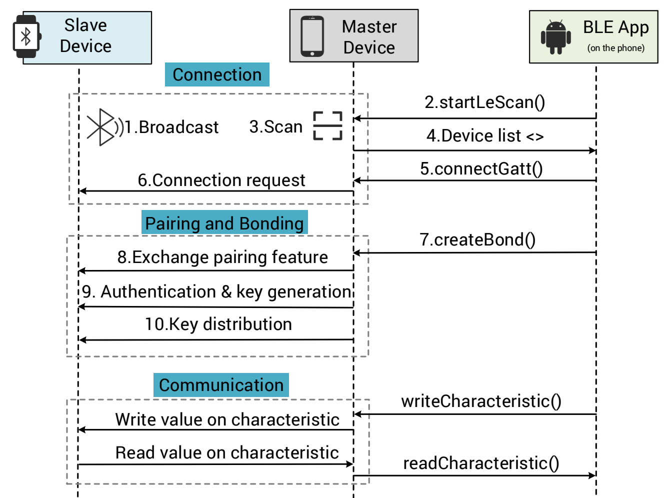

Fig. 1: BLE workflow with a mobile app.

Then the smartphone running in the scan mode can receive the advertising packets and further request additional data from the device. Based on the received data, if the smartphone is interested in this device, it will send a connect request to the device. After the above processes, the connection between the phone and the device will be established.

2) Pairing and Bonding Stage. Pairing and bonding is a mechanism to achieve secure communication, which has two security goals: protection against passive eavesdropping and active man-in-the-middle (MITM) attacks. Initially, the smartphone and IoT device exchanged their pairing features (I/O capabilities and security requirements). Further, these features are used to determine the concrete pairing method.

BLE supports four pairing methods, including *Just Works*, *Passkey Entry*, *Numeric Comparison*, and *Out of Band (OOB)*. *Just Works* is the weakest pairing method, suffering the risk of MITM attacks [8]. Only the *Passkey Entry* and *Numeric Comparison* methods can defend against MITM attacks. Note that the security of *OOB* method depends on how it is implemented, such as via QR code or NFC channel. Besides, the I/O capabilities also affect the method selection. For example, *Passkey Entry* requires the user to enter a 6-digit passcode from the display of the other device. *Numeric Comparison* requires the user to confirm that the same 6-digit passcode is displayed on both devices. If either device has no I/O capabilities (e.g., display or keyboard), the *Just Works* method has to be used.

After selecting the pairing method, the smartphone and IoT device negotiate a long-term key (LTK) to encrypt the communication channel in the pairing stage. Then, the Identity Resolving Key (IRK) and Connection Signature Resolving Key (CSRK) are generated from one device and distributed to the other. CSRK is used for signature generation/verification, and IRK for identity resolution. In the bonding stage, LTK will be stored in two devices for further communication.

Authorized licensed use limited to: SHANDONG UNIVERSITY. Downloaded on November 02,2022 at 14:14:45 UTC from IEEE Xplore. Restrictions apply.

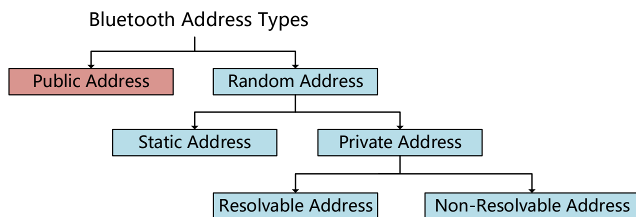

Fig. 2: Types of Bluetooth MAC address.

3) Communication Stage. Now, the smartphone and the IoT device can exchange data. The basic data unit for reading and writing in BLE is called *attribute*, and its format follows the Generic Attribute Profile (GATT). An attribute involves four properties: handle, UUID, value, and permission. A UUID is a 128-bit number used to identify a specific BLE attribute, including service, characteristic, and descriptor.

B. Bluetooth MAC Address

There are two types of Bluetooth MAC addresses: *public address* and *random address*, as shown in Figure 2. A Bluetooth public address is a fixed global address that must be registered with IEEE [12]. The random address contains two sub-types – *static address* and *private address*. The random static address can be fixed throughout the device's lifespan or changed only when rebooted. The random private address is specially used for the privacy protection of a Bluetooth device by hiding the identity and averting the device tracking risk. The random private addresses can be resolvable or non-resolvable, as introduced below.

- *Resolvable Random Private Address* – This kind of address is "resolvable" by using IRK shared with a particular trusted device. Usually, we call it the dynamic address generation technique.
- *Non-Resolvable Random Private Address* – Its main difference from resolvable addresses is that any other device cannot resolve it. This type is uncommon and is only used in non-connectable mode (e.g., beacons).

III. Problem Overview

In this section, we first summarize four typical attacks against BLE-enabled devices. By analyzing the causes of the attacks, we then identify five common BLE misconfigurations on the devices.

A. BLE Attacks

Threat Model. In this work, we focus on the smart IoT devices which use BLE to communicate with their companion mobile apps. Under this scenario, we assume the attackers can sniff all Bluetooth packets transmitted over the air. For example, the attacker can use a Bluetooth dongle, such as the CC2540 sniffer from Texas Instrument [9], to capture packets broadcast and transmitted from all IoT devices or a particular device within a reasonable distance.

Attack Categories. Based on the attackers' capabilities, the main security threats caused by BLE misconfigurations are as follows.

A1) Eavesdropping. The attacker can monitor the unencrypted BLE communications to steal data as it is being sent or received by the user.

A2) Identity Tracking. The attacker can keep track of the IoT device through the identity information broadcast by the device. There are two types of identity information that can be abused for tracking – static Bluetooth MAC address and static UUID. When broadcasting, a BLE device will notify its MAC address for the peer to connect with it. If the MAC address is static, it will open up the possibility of an identity tracking attack [8]. Also, after the mobile phone initiates a connection to the IoT device, the device will return the UUIDs of its available services to the phone. Similarly, the static UUIDs also can lead to an identity tracking attack.

A3) Man in the Middle (MITM). Except for the passive eavesdropping, the attacker can maintain two separate connections with each BLE entity and relay their messages in the middle. In this case, the entire communication session is under the attacker's control (i.e., MITM attack). Before the attack, if the connection has already been established, the attacker may need to use a signal jammer to trigger a re-connection by preventing the data broadcasting.

A4) Unauthorized Access. Unauthorized access allows attackers to read/write data or issue commands to IoT devices. Note that a secure pairing method cannot authenticate the current user's identity. The attacker can still physically access the deployed IoT device and complete the pairing. If the IoT devices lack the application layer authentication, they cannot verify the entity's identity that sends the control command, resulting in unauthorized access.

B. BLE Security Misconfigurations

The above security threats can be avoided if the developers comply with Bluetooth specifications and configure the BLE devices correctly. On the contrary, misconfigurations may be introduced by developers. To investigate the existing security risks and measure the affected scope, this paper proposes five questions on misconfigurations derived from the above BLE attacks (the associated attack is marked within each question).

Q1) Does the device use a static Bluetooth MAC address? **(*A2*)** As mentioned before, the static MAC address of a BLE device can be leveraged to identify that device uniquely. The dynamic MAC address generation technique should be adopted to mitigate such risk.

Q2) Does the device use static UUIDs? **(*A2*)** The UUIDs are used initially to recognize BLE attributes, such as BLE services and characteristics. They can be retrieved from the advertising data or communication data. The recent research [31] showed that the static UUIDs could enable an attacker to precisely fingerprint an IoT device. The correct configuration is to generate UUIDs dynamically or use encrypted UUIDs.

Authorized licensed use limited to: SHANDONG UNIVERSITY. Downloaded on November 02,2022 at 14:14:45 UTC from IEEE Xplore. Restrictions apply.

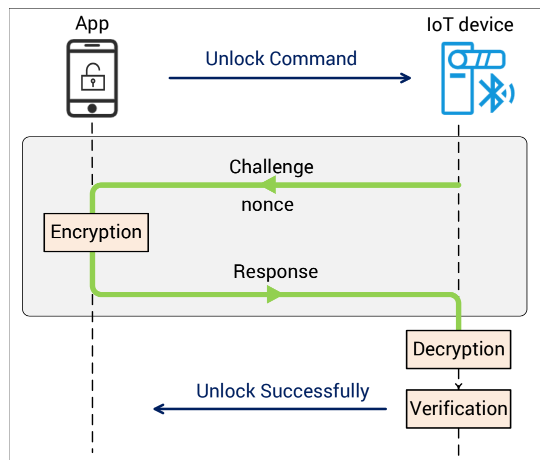

Fig. 3: Challenge-response authentication.

Q3) Does the device use an insecure pairing method? **(*A3*)** *Just Works* is the weakest pairing method and can not prevent the MITM attacks. The developers should use the other methods – *Passkey Entry*, *Numeric Comparison*, or *OOB*.

Q4) Does the device use plaintext for data transmission? **(*A1*)** In the BLE link layer, the communication channel will be encrypted after the pairing stage. In the application layer, the mobile app can achieve proprietary key agreement with IoT devices and further use the secret key to encrypt their communication. If neither of the approaches is taken, resulting in plaintext transmission, the attacker is able to eavesdrop on the BLE communication.

Q5) Does the device use the application layer authentication? **(*A4*)** Although the BLE pairing enforces device authentication against MITM attacks, it also has the limitation of user authentication, leading to unauthorized access. The developers should deploy the application-layer authentication for access control, such as the challenge-response authentication protocol as shown in Figure 3. This protocol usually employs a cryptographic nonce as the challenge to ensure that every challenge-response sequence is unique, which is against eavesdropping with a subsequent replay attack [10]. First, the IoT device sends a challenge (nonce) to its companion app. The app processes this challenge to generate an answer and sends it back to the IoT device. Finally, the IoT device verifies this answer to ensure the app's identity.

IV. Design of BSC-Checker

In this section, we first present the high-level idea of our analysis framework – BSC-Checker (**B**LE **S**ecurity **C**onfiguration Checker). We then introduce the detailed detection strategies to identify the BLE security misconfigurations in companion mobile apps.

Motivation. In order to discover vulnerabilities in BLE-enabled IoT devices, one way is to analyze the device firmware. However, due to the difficulties in obtaining and investigating firmware, large-scale analysis of IoT devices is quite challenging. For example, even if we could successfully

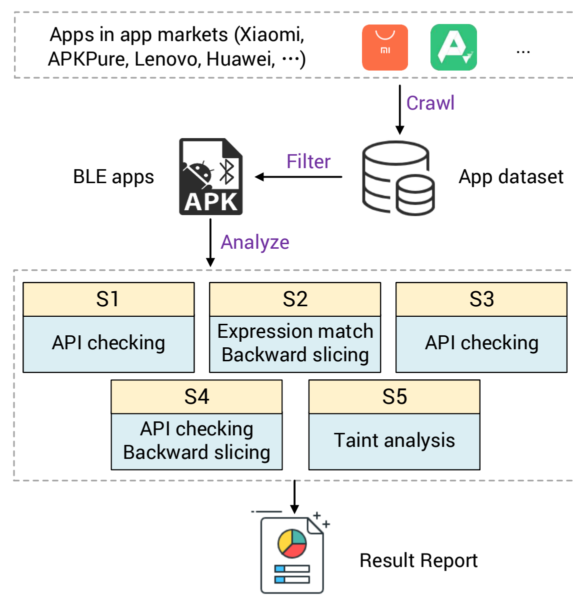

Fig. 4: Analysis framework.

extract and reverse-engineer the device firmware, the Bluetooth driver and application-layer BLE APIs may vary across different vendors.

By contrast, we found it more feasible and efficient to analyze the companion mobile apps that are usually responsible for interacting with the IoT devices. Android (the leading mobile OS) system provides built-in support for BLE [2]. Apps can conveniently discover BLE devices, query for services, and transmit data through unified APIs. Meanwhile, Android apps can be massively obtained from mainstream app markets. More importantly, the companion apps must share the same set of configurations with the corresponding devices and will reflect the devices' security states. For instance, the app can only use an identical MAC address and UUIDs on the device side for communication. Any application layer encryption or authentication implementation on the device will have its counterpart on the app side. Therefore, we can effectively measure the device misconfigurations (questions in Section III-B) through a large-scale analysis of the companion apps.

A. Analysis Workflow

In our framework – BSC-Checker, to detect the misconfiguration issues, we first need to obtain the companion apps of BLE-enabled IoT devices. Then we evaluate each misconfiguration case on the collected BLE apps. In this process, we applied different program analysis techniques (such as taint analysis and backward slicing) with our detection strategies. Figure 4 shows the overall workflow of BSC-Checker. In summary, the main steps include:

1) *App Pre-processing.* First, we construct an app dataset and then filter out the apps related to BLE.
2) *Strategy Checking.* Next, we input the collected BLE app dataset into the BSC-Checker detection module and analyze each app based on our detection strategies.

Authorized licensed use limited to: SHANDONG UNIVERSITY. Downloaded on November 02,2022 at 14:14:45 UTC from IEEE Xplore. Restrictions apply.

3) *Report Output.* In the last step, the framework outputs a report that indicates the misconfigurations in IoT devices based on the detection results of companion apps.

App Pre-processing. After collecting massive apps from app markets, we first filter out the apps that declare Bluetooth permissions (i.e., `BLUETOOTH` and `BLUETOOTH_ADMIN`). Since the app filtered by Bluetooth permissions may include Bluetooth Classic and BLE simultaneously, we then use the BLE-related classes (e.g., `android.bluetooth.BluetoothGatt`) as a condition to further filter out BLE apps. Note that the permission checking relies on the app `Manifest` files without the need to disassemble code, which significantly speeds up the process.

Strategy Checking. Given a group of Android BLE apps, BSC-Checker uses a toolset to perform basic analysis on each app. Our approach targets the Dalvik bytecode of Android apps directly. We build our analysis framework on top of Androguard [1] and FlowDroid [14]. Androguard is an open-sourced static analysis tool. BSC-Checker utilizes it to decompile an app into classes, methods, basic blocks, and individual instructions. FlowDroid is an open-sourced taint analysis framework. BSC-Checker incorporates it to construct call graph, control graph, and perform taint analysis on apps.

Once the above basic analysis is done, BSC-Checker uses these preliminary results to detect the misconfigurations according to our proposed detection strategies. We carefully design each detection strategy in order to answer each *question* (in Section III-B). Our detection tool will locate the key code snippet that reveals the possible BLE misconfigurations. We elaborate the detection methods in detail in Section IV-B.

Report Output. The misconfigurations found from the app imply that they are also present in the IoT devices. BSC-Checker outputs a security report based on these misconfiguration results. The report reveals the security threats to real IoT devices, such as identity tracking and MITM risks.

B. Detection Strategies

To answer the questions in Section III-B, we present our detection solutions in detail.

***S1)* Does the device use a static Bluetooth MAC address?** As stated in Section II-B, both *public address* and *static address* can be tracked. The *non-resolvable address* is only adopted by unconnectable devices. So the *resolvable address* (could be changed dynamically) should be used to defend against identity tracking attacks. When this MAC address is changed, the peer device could use the IRK to resolve the updated one. The IRK is only negotiated and distributed in the pairing and bonding stage. Thus, we conclude that if an IoT device and a smartphone skip this stage when establishing a connection, it must use a static MAC address. The API `createBond()` is the only Android method that a BLE app can invoke for initiating the pairing process. Therefore, our strategy is to scan and check whether this API is invoked in the BLE app.

***S2)* Does the device use static UUIDs?** In principle, an IoT device using static UUIDs is subject to the same identity tracking attack as using a static MAC address [16] [31]. These static UUIDs are hard-coded in the firmware of the IoT device. The developers often hard-code these UUIDs in the companion apps for convenience as well. Thus, we consider that if the app tries to read and write data using hard-coded UUIDs, the corresponding IoT device must be using the same static UUIDs. According to the BLE specification, the UUIDs are typically 128-bit hexadecimal strings. In our detection, we first extract all static UUID strings by regular expressions matching over the app. But not all UUIDs are used in the BLE communication (e.g., some Java objects in Android also have the same formatted UUIDs). In order to determine whether the static UUIDs are used in BLE, we apply the backward slicing technique to locate the BLE methods that take these UUIDs as parameters. If the UUIDs are hard-coded and invoked by a BLE method, we conclude that the IoT device is vulnerable to identity tracking (fingerprinting) attack.

***S3)* Does the device use an insecure pairing method?** As mentioned in Section II, BLE provides four pairing methods. Except for OOB (need to manually specify), one of the other three methods will be automatically selected based on the I/O capabilities of the two devices. Although we intuitively believe that OOB is a rarely-used pairing method, we still scanned the usage of such a method to ensure the accuracy of the detection. Android has a hidden API `createoutofband()` for the OOB pairing method. So we first detect whether the app uses the Java reflection technique to invoke this API. If not, the pairing method will be chosen from *Passkey Entry*, *Numeric Comparison*, and *Just Work*.

To determine which remaining method will be used, we observed that if an IoT device requires MITM protection, either *Passkey Entry* or *Numeric Comparison* pairing methods will be enabled according to its I/O capability. For *Passkey entry*, Android provides the API `setpin()` for prompting the user to enter the PIN (e.g., `123456`). When the API `setConfirmation()` is called within the app, the user will be prompted to confirm the passkey displayed on the screen. If neither API is detected, then the *Justwork* method will be used by default.

***S4)* Does the device use plaintext for transmission?** There are two ways to encrypt the communication between an IoT device and its companion app to prevent eavesdropping attacks. One is the link layer encryption supported by the underlying Bluetooth protocol, and the other is the application layer encryption that is customized by the device firmware and its companion app.

Link Layer. According to the Bluetooth specification, the link layer encryption of BLE is optional. For example, when an IoT device acts as a GATT server, it can specify its attribute permission as Encryption. In this case, if the companion app acts as a GATT client to read/write that attribute of the device, the link layer of this communication will be encrypted. To achieve this, the device and app must establish the pairing and

Authorized licensed use limited to: SHANDONG UNIVERSITY. Downloaded on November 02,2022 at 14:14:45 UTC from IEEE Xplore. Restrictions apply.

bonding to negotiate the encryption key. So in the detection, we check whether the corresponding API `createBond()` is invoked in the app.

Although Android encapsulates the link layer encryption functionality into a single API, it also states the risk of using such method. The Android announcement says "*When a user pairs their device with another device using BLE, the data that's communicated between the two devices is accessible to all apps on the user's device*" [6]. Therefore, Android also recommends application layer encryption for BLE communication.

Application Layer. The Bluetooth chip manufacturers (e.g., Nordic) usually provide their SDKs that include various crypto libraries. In the firmware, the device developers can use these crypto APIs to customize their own encryption schemes. For BLE apps, Android provides the Java crypto libraries (e.g., `java.security` and `javax.crypto`) for data encryption. Sivakumaran et al. [22] developed a tool called BLECryptracer for detecting BLE application layer security issues. To detect whether an IoT device is using application layer encryption, our solution is as follows.

In the IoT device firmware, data is encrypted by crypto APIs and transmitted to the companion app via BLE. The app must decrypt the received message to restore the plain data. Thus, we could trace the data related to the BLE message sending/receiving API (e.g., `writeValue()` and `getValue()`) and check whether the data is encrypted/decrypted in the process. Initially, we consider tainting this data flow, which makes the encryption method as the "source" and the BLE message sending API as the "sink" to detect the existence of such a path in the app. We compared the testing results with BLECryptracer [22] and found that BLECryptracer is a bit more accurate. The reason is that FlowDroid cannot process the apps with native code and large size, causing a high failure rate of the path-building. On the contrary, BLECryptracer adopts a lightweight technical solution based on backward slicing. Therefore, finally, we conducted our detection based on BLECryptracer.

We first filtered out the apps with the objective method. BSC-Checker is optimized based on BLECryptracer so that it only detects the links between BLE and cryptographic functions through direct register value transfers and direct results of method calls. We eliminate the coarse-grained analysis method in BLECryptracer, which could lead to false positives.

S5) Does the device use application layer authentication? In a public environment, a BLE-enabled IoT device can interact with multiple users. The BLE standard does not enforce the requirement that only users with permissions can interact with BLE devices. To address the unauthorized access problem, an application layer authentication is necessary. Since there is no uniform standard for the authentication protocol, each device vendor could implement it in different ways [23], which inevitably makes detection difficult.

As mentioned in Section III, we mainly focus on the commonly deployed challenge-response protocol. Specifically, the IoT device sends a challenge (nonce) to its companion app, and this app will reply with an answer to the device. A sample app code is shown in Listing 1. Our detection method uses the "challenge" sent from the IoT device as the taint source, the "challenge" processing by the app as taint propagation, and sending the "response" to the IoT device as the taint sink. So the target APIs we want to monitor for receiving challenges are `readCharacteristic()` and `onCharacteristicChanged()` , and the target API for sending response is `writeCharacteristic()`. If FlowDroid does not find the tainted path, we consider that the challenge-response authentication protocol does not exist between the IoT device and its companion app.

```
1  public void onResponse(Chanllenge ch) {
2    bAuth = ch.getIntValue(20,0);
3    Response re = authService.
         getAuthVectorCharacteristic();
4    re.setValue(AuthMath.secondStepSecret(
         aAuth, bAuth, macAddress, key));
5    blueRock.writeCharacteristic(re, new
         OperationCallback(){
6      public void onSuccess() {
7        BeaconAuthentication.onCompleted();
8      }
9      public void onFailure() {
10       L.w("Authentication failed");
11       BeaconAuthentication.onFailed();
12     }
13   }
14
15 public static byte[] secondStepSecret(long
     aAuth, long bAuth, MacAddress macAddress
     , byte[] key) {
16   try {
17     return aesDecrypt(sessionKey(aAuth,
         bAuth), aesEncrypt(key,
         macAddressToMacSecret(macAddress)));
18   } catch (Exception e) {
19     return null;
20 }
```

Listing 1: Code example of challenge and response.

V. Results and Findings

We implemented the prototype of our analysis framework BSC-Checker and conducted large-scale evaluations. Here we summarized our findings.

A. Experiment Setup and Dataset

Since there is no available large-scale Android BLE app dataset, we developed a crawler to download the latest Android apps from popular app markets[1]. We selected 18 of them, including 9Apps, 2265, Anzhi, APKPure, Baidu, Download-PCAPK, FDroid, Gfan, Huawei, LapTopPCAPK, Lenovo, LePlay, Leyou, Flyme, PC6, Yingyongbao, Uptodown, and Xiaomi. Finally, we collected 226,181 APKs from the above app markets. To filter BLE-based apps among the collected APKs,

[1]Google Play did not support the app bulk downloading, we did not consider it in our experiment.

Authorized licensed use limited to: SHANDONG UNIVERSITY. Downloaded on November 02,2022 at 14:14:45 UTC from IEEE Xplore. Restrictions apply.

TABLE I: Overall results.

Questions	Item	Value	Percentage
Q1) Static Bluetooth MAC address identification	#Static MAC address identified	4002	87.2%
Q2) Static UUIDs identification	#Static UUIDs identified	3954	86.2%
Q3) Insecure paring method identification	#Just Works pairing method	298	6.5%
Q4) Plaintext transmission identification	#Plaintext transmission identified	3960	86.3%
Q5) Application layer authentication identification	#Application layer authentication	20	0.4%

we first filtered out 35,702 apps that use regular Bluetooth permissions (i.e., `BLUETOOTH` and `BLUETOOTH_ADMIN`). Among them, we further obtained 4,589 BLE apps.

Environment setup. Our evaluation consists of two sets of experiments: the static analysis of BLE apps and attacks against real-world BLE-enabled IoT devices. The static analysis was conducted on a Linux server running Ubuntu 20.04 equipped with Intel Xeon 6226R @2.90GHz and 256G memory. The attacks were conducted by Raspberry Pi 4 and CC2540 BLE USB Dongle.

B. Overall Results

We deployed BSC-Checker on these 4,589 BLE apps to answer the research questions discussed in Section III, and the final results are summarized in Table I.

***R1)* Static Bluetooth MAC address identification.** BSC-Checker discovered that most BLE apps (87.2%) used static MAC addresses. We conducted a deeper analysis of these misconfigurations. The developers store the MAC address in their apps' cache after the first successful connection for convenience, which exposes that the MAC addresses of IoT devices are constant.

***R2)* Static UUIDs identification.** BSC-Checker found 86.2% of BLE apps used static UUIDs. Since the UUID-based tracking attack was proposed in recent two years, many developers may not be concerned about this issue. The correct approach is not to hardcode the UUIDs in the app but use dynamic UUIDs in the IoT device.

***R3)* Insecure pairing method identification.** BSC-Checker detected 588 apps (12.8%) using the pairing and bonding methods. Among them, only 1 app uses the OOB paring method, which is the least. There are 289 apps using secure pairing methods (*Passkey Entry* or *Numeric Comparison*). Up to half of them (298 apps) use the insecure pairing method (*Just Works*). The corresponding IoT devices that use insecure pairing methods may be due to their limited I/O capabilities.

***R4)* Plaintext transmission identification.** BSC-Checker found 4,002 apps that did not use Bluetooth link layer encryption. Among them, only 42 apps used the application layer encryption. That is, 3,960 apps (86.3%) transmit plaintext messages. For those IoT devices that use plaintext transmission, they may be energy-sensitive devices.

***R5)* Application layer authentication identification.** BSC-Checker found that only a small portion of them (0.4%) used application layer authentication. The corresponding devices may have high-security requirements, such as smart locks.

C. Case Studies

In this subsection, we provide two cases to demonstrate the effectiveness of BSC-Checker. The first one is to verify the discovered app vulnerability in the device firmware. The second one is to launch practical attacks exploiting the identified vulnerability.

Verification in firmware. The first challenge is to obtain the firmware. Many IoT devices support OTA (over-the-air) services that allow regular firmware updates. The most common way is to download the latest firmware from the vendor's server via the Internet. However, some device vendors do not maintain the server for updating services due to cost or technical issues. Instead, their method is to package the new firmware in their device companion apps and publish the apps to app markets. When the IoT device user updates the companion app, the new firmware will be transmitted to the device for updating. Therefore, we can unpack apps to find the firmware-related binaries. Due to Nordic's high market share and detailed development documentation, our goal was to find the firmware used for Nordic Bluetooth chips. The basic approach is to match the features (e.g., magic number) in the binaries related to the Nordic firmware.

In this case, we analyzed the app Findster (used for looking for lost pets) – `com.getfindster.android.findsterduo`. The report output by BSC-Checker shows that the corresponding device is vulnerable to eavesdropping, identity tracking, MITM, and unauthorized access attacks.

Then, we decompiled the extracted firmware using Binary Ninja [4]. The Nordic firmware invokes the APIs related to the BLE protocol stack through SVC interrupts [3]. Each API corresponds to an SVC exception number (e.g., `SVC 0x7D`). We used FirmXray [25] to analyze the firmware for Bluetooth link layer security vulnerabilities. According to its detection report, we identified the static MAC addresses through the first target API `sd_ble_gap_addr_set`. The second target API `sd_ble_gatts_service_add` is used to generate the GATT service, which contains the UUIDs. Through the third API `sd_ble_gap_sec_params_reply`, we can determine which pairing method is used for IoT devices. The last API `sd_ble_gatts_characteristic` allows us to determine whether the IoT device needs the link layer encryption.

Based on the results, we determined that the corresponding IoT device used static MAC addresses and static UUIDs. No encryption and pairing methods are used at the BLE link layer. We need further to check the application layer encryption and authentication protocol. Nordic provides AES encryption API `sd_ecb_block_encrypt` in the SDK [11]. The encryption API's exception number is `0x48`. Therefore, we looked for the instruction `SVC 0x48` in the firmware to determine whether the application-layer encryption is being used. The results show that this firmware transmits plaintext messages with its app. As for authentication, this firmware uses application-layer authentication to generate

Authorized licensed use limited to: SHANDONG UNIVERSITY. Downloaded on November 02,2022 at 14:14:45 UTC from IEEE Xplore. Restrictions apply.

TABLE II: Attacks on IoT devices.

Device	Questions					Attacks			
	Q1	Q2	Q3	Q4	Q5	A1	A2	A3	A4
Smart Bulb	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
Smart Lock	Yes	Yes	Yes	No	Yes	No	Yes	Yes	No

a nonce as a challenge. Nordic provides a nonce generation API `sd_rand_application_bytes_available_get()`. We did not find the use of this API in the firmware. Combined with the previous result of not using the application layer encryption API, it suggests that this firmware does not use application-layer authentication.

Attack on IoT devices. To confirm the vulnerabilities revealed by BSC-Checker, we purchased 2 devices (LifeSmart smart bulb LS030UN and OKLOK smart lock BL-80) to perform practical attacks, as shown in Table II. To launch attacks, we built an attacking device based on a Raspberry Pi 4 and a CC2540 BLE USB Dongle.

First, we used the dongle to sniff the advertising packets broadcast by the devices. To Q1 & Q2: we saved the MAC addresses and UUIDs obtained from the broadcast packets, rebooted the device, and repeated this step. The result revealed that these data did not change. To Q3: we then connected to the devices and found that the companion apps did not prompt pairing requests. To Q4: we hooked the critical methods in the apps and compared the messages before and after sending, and found that the smart bulb app (`com.ilifesmart.mslict_gp`) uses plaintext transmission and the smart lock app (`com.oklok.lock`) uses ciphertext transmission. To Q5: we used Raspberry Pi 4 to connect the device and replay the control commands. The bulb app responded correctly to the replayed command, but the lock app did not. The report output by BSC-Checker shows that the lock does not deploy the application layer authentication. However, in the attack, it sent challenges (nonce). We further reverse-engineered this app and found it used web-based techniques (JavaScript) to implement the application layer authentication. Therefore, it led to a false positive in our detection because our tool only detects Java code.

VI. Discussions

In this section, we discussed some limitations of this work.

Unavailable Code. Our BLE app analysis is based on the well-established Flowdroid and Androguard. However, some APK files cannot be unpacked or disassembled by them successfully, which further affects the following analysis. It is mainly caused by the code obfuscation and APK packing. In addition, some apps implement certain processing logic in native code (in the format of `.so` files) or web code, and our framework only can handle Java/smali code.

App to Device. The discovered misconfigurations in BLE apps can be the clues to identifying the same problems in BLE-enabled devices. However, it is not easy to map the app to its belonged device in some cases due to lacking product information. Also, an app may correspond to multiple devices, and not all device firmware exists the discovered problems.

VII. Related Work

This section reviews the previous works about BLE security, focusing on BLE vulnerability, privacy leakage, and automated security analysis.

BLE Vulnerabilities. Since BLE has been widely applied for many years, its vulnerabilities and corresponding defense measures have been well discussed. For example, Zhang et al. [29] presented severe flaws of Secure Connections Only (SCO) mode in case of improper handling of the BLE programming framework of the initiator. It will result in that a fake device can perform downgrade attacks to steal users' sensitive data by spoofing a victim BLE device through the BLE pairing protocol. Tschirschnitz et al. [24] revealed a design flaw in the inconsistent association model in the BLE pairing process and taking advantage of this fact, the method confusion attack can easily achieve a MITM position of two BLE devices. Wu et al. [27] aimed at the reconnection procedure targeting the BLE link-layer authentication mechanism. They proposed a BLE spoofing attack in which an attacker can provide spoofed data to a previously-paired BLE client device by pretending to be a BLE server device. Other related works include reflection attack [17], profile management flaw [28], key negotiation downgrade attack [13], and malicious traffic injection [15].

BLE Privacy Leakage. The BLE-based IoT solutions enable devices to exchange information efficiently in real-time. Nevertheless, the potential leakage can also cause severe harm to users' privacy (e.g., user tracking and behavior monitoring) [18], [20]. In the defense, Fawaz et al. [19] proposed a device-agnostic system called BLE-Guardian. It determines the entities that can observe the device's existence through the advertisements to ensure that only user-authorized entities can connect to those BLE-equipped devices. However, it cannot protect devices from spoofing attacks. Wu et al. [26] proposed BlueShield utilizing the BLE device identity information carried by advertising packets to filter out malicious packets from an attacker, which better defend against spoofing attacks.

Automated Analysis. Some recent research concentrated on designing automated analysis tools to discover BLE-related bugs. Zuo et al. [31] performed an automatic tool BleScope to scan BLE device's vulnerabilities through its companion mobile apps. A design flaw of its present implementation enables an attacker to fingerprint a BLE device with static UUIDs from the apps, causing the security risks of communication between these victim devices and their companion apps. Sivakumaran et al. [22] introduced a static analysis tool called BLECryptracer to investigate the application-layer security of 18,900+ BLE-enabled Android apps. Their result shows that over 45% collected apps do not protect BLE data well. Ray et al. [21] developed a framework that allows running MITM attacks, flooding attacks, and fuzzing for BLE devices.

The closest works with our study were FirmXRay [25] and BLESS [30]. Wen et al. [25] proposed a static analysis tool FirmXRay to detect BLE link layer vulnerabilities. They analyzed static MAC address, Just Works pairing, and key exchange from the configurations of bare-metal firmware.

Authorized licensed use limited to: SHANDONG UNIVERSITY. Downloaded on November 02,2022 at 14:14:45 UTC from IEEE Xplore. Restrictions apply.

However, the correctness of firmware disassembling limits this work, and not all of these three types of analysis are used. On the contrary, our solution focuses the mirror of firmware – BLE apps, and there is no firmware needed in the analysis process. Zhang et al. [30] proposed a BLE app security scanning tool (BLESS) to check the security practices of BLE products. The checking focuses on the use of cryptographic keys and nonces. Compared with this solution, we focus on the BLE misconfiguration problems covering more detection strategies. Also, our experiment scale is more extensive than this prior study, say 1,073 apps vs. 4,589 apps, which provides enough data to support the misconfiguration measurement.

VIII. Conclusion

This paper studied the BLE security misconfiguration problems in the IoT ecosystem. Specifically, we went through the official documents and summarized five checking strategies. Also, to avoid the challenges of device firmware analysis, our solution tries to analyze the mirror of firmware – companion mobile apps. We designed an automated analysis tool – BSC-Checker, which can generate the configuration checking results for the input apps. In the large-scale experiment, the result shows that the security configurations of most BLE apps disobey at least one security rule. Accordingly, most BLE deployments in the wild are risky.

Acknowledgements

We thank the anonymous reviewers for their insightful comments. This work was partially supported by National Natural Science Foundation of China (No. 61902148), Hong Kong S.A.R. Research Grants Council (RGC) General Research Fund (No. 14208019), and Shandong Provincial Natural Science Foundation (No. ZR2020MF055, No. ZR2021LZH007, No. ZR2020LZH002, and No. ZR2020QF045).

References

[1] Androguard. https://github.com/androguard/androguard.
[2] Android Bluetooth APIs – android.bluetooth. https://developer.android.com/reference/android/bluetooth/package-summary.
[3] ARM Instruction sets. https://developer.arm.com/architectures/instruction-sets.
[4] Binary Ninja. https://binary.ninja/.
[5] Bluetooth. https://www.bluetooth.com/.
[6] Bluetooth Low Energy. https://developer.android.com/guide/topics/connectivity/bluetooth/ble-overview.
[7] Bluetooth low energy (BLE) enabled devices market volume worldwide, from 2013 to 2020. https://www.statista.com/statistics/750569/worldwide-bluetooth-low-energy-device-market-volume/.
[8] Bluetooth Specification Version 4.2. https://www.bluetooth.com/specifications/specs/core-specification-4-2/.
[9] CC2540 – Bluetooth Low Energy wireless MCU with USB. https://www.ti.com/product/CC2540.
[10] Challenge-Response Authentication. https://en.wikipedia.org/wiki/Challenge-response_authentication.
[11] Nordic SDK. https://infocenter.nordicsemi.com/index.jsp.
[12] M. Afaneh. Bluetooth Addresses & Privacy in Bluetooth Low Energy. https://www.novelbits.io/bluetooth-address-privacy-ble/.
[13] D. Antonioli, N. O. Tippenhauer, and K. Rasmussen, "Key Negotiation Downgrade Attacks on Bluetooth and Bluetooth Low Energy," *ACM Transactions on Privacy and Security (TOPS)*, vol. 23, no. 3, pp. 14:1–14:28, 2020.
[14] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. D. McDaniel, "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps," in *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Edinburgh, United Kingdom, June 9-11, 2014*, 2014.
[15] R. Cayre, F. Galtier, G. Auriol, V. Nicomette, M. Kaâniche, and G. Marconato, "InjectaBLE: Injecting Malicious Traffic into Established Bluetooth Low Energy Connections," in *Proceedings of the 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Taipei, Taiwan, June 21-24, 2021*, 2021.
[16] G. Celosia and M. Cunche, "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 (IoT S&P@CCS), London,UK, November 15, 2019*, 2019.
[17] T. Claverie and J. Lopes-Esteves, "Bluemirror: Reflections on bluetooth pairing and provisioning protocols," in *Proceedings of the 15th IEEE Workshop on Offensive Technologies (WOOT), San Francisco, CA, USA, May 27, 2021*, 2021.
[18] A. K. Das, P. H. Pathak, C. Chuah, and P. Mohapatra, "Uncovering Privacy Leakage in BLE Network Traffic of Wearable Fitness Trackers," in *Proceedings of the 17th International Workshop on Mobile Computing Systems and Applications (HotMobile), St. Augustine, FL, USA, February 23-24, 2016*, 2016.
[19] K. Fawaz, K. Kim, and K. G. Shin, "Protecting Privacy of BLE Device Users," in *Proceedings of the 25th USENIX Security Symposium (USENIX-Sec), Austin, TX, USA, August 10-12, 2016*, 2016.
[20] A. Korolova and V. Sharma, "Cross-App Tracking via Nearby Bluetooth Low Energy Devices," in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy (CODASPY), Tempe, AZ, USA, March 19-21, 2018*, 2018.
[21] A. Ray, V. Raj, M. Oriol, A. Monot, and S. Obermeier, "Bluetooth low energy devices security testing framework," in *Proceedings of the 11th IEEE International Conference on Software Testing, Verification and Validation (ICST), Västerås, Sweden, April 9-13, 2018*, 2018.
[22] P. Sivakumaran and J. Blasco, "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 *Proceedings of the 28th USENIX Security Symposium (USENIX-Sec), Santa Clara, CA, USA, August 14-16, 2019*, 2019.
[23] D. Veilleux. (2017) Simple Application-level Authentication. https://devzone.nordicsemi.com/nordic/nordic-blog/b/blog/posts/simple-application-level-authentication.
[24] M. von Tschirschnitz, L. Peuckert, F. Franzen, and J. Grossklags, "Method Confusion Attack on Bluetooth Pairing," in *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland), San Francisco, CA, USA, 24-27 May 2021*, 2021.
[25] H. Wen, Z. Lin, and Y. Zhang, "FirmXRay: Detecting Bluetooth Link Layer Vulnerabilities From Bare-Metal Firmware," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS), Virtual Event, USA, November 9-13, 2020*, 2020.
[26] J. Wu, Y. Nan, V. Kumar, M. Payer, and D. Xu, "BlueShield: Detecting Spoofing Attacks in Bluetooth Low Energy Networks," in *Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID), San Sebastian, Spain, October 14-15, 2020*, 2020.
[27] J. Wu, Y. Nan, V. Kumar, D. J. Tian, A. Bianchi, M. Payer, and D. Xu, "BLESA: Spoofing Attacks against Reconnections in Bluetooth Low Energy," in *Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT), August 11, 2020*, 2020.
[28] F. Xu, W. Diao, Z. Li, J. Chen, and K. Zhang, "BadBluetooth: Breaking Android Security Mechanisms via Malicious Bluetooth Peripherals," in *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS), San Diego, California, USA, February 24-27, 2019*, 2019.
[29] Y. Zhang, J. Weng, R. Dey, Y. Jin, Z. Lin, and X. Fu, "Breaking Secure Pairing of Bluetooth Low Energy Using Downgrade Attacks," in *Proceedings of the 29th USENIX Security Symposium (USENIX-Sec), August 12-14, 2020*, 2020.
[30] Y. Zhang, J. Weng, Z. Ling, B. Pearson, and X. Fu, "BLESS: A BLE Application Security Scanning Framework," in *Proceedings of the 39th IEEE Conference on Computer Communications (INFOCOM), Toronto, ON, Canada, July 6-9, 2020*, 2020.
[31] C. Zuo, H. Wen, Z. Lin, and Y. Zhang, "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 (CCS), London, UK, November 11-15, 2019*, 2019.

Authorized licensed use limited to: SHANDONG UNIVERSITY. Downloaded on November 02,2022 at 14:14:45 UTC from IEEE Xplore. Restrictions apply.