BLESS: A **BLE** Application **S**ecurity **S**canning Framework

Yue Zhang[1,2], Jian Weng[1,*], Zhen Ling[4], Bryan Pearson[2], and Xinwen Fu[2,3]

[1]College of Information Science and Technology, Jinan University,
Guangzhou, Guangdong,China. Email: zyueinfosec@gmail.com, cryptjweng@gmail.com
[2]Department of Computer Science, University of Central Florida, FL, USA.
Email: yue.zhang@ucf.edu, bpearson@knights.ucf.edu, xinwenfu@ucf.edu
[3]Department of Computer Science, University of Massachusetts Lowell, MA, USA.
[4]School of Computer Science and Engineering, Southeast University,China. Email:zhenling@seu.edu.cn

***Abstract*—Bluetooth Low Energy (BLE) is a widely adopted wireless communication technology in the Internet of Things (IoT). BLE offers secure communication through a set of pairing strategies. However, these pairing strategies are obsolete in the context of IoT. The security of BLE based devices relies on physical security, but a BLE enabled IoT device may be deployed in a public environment without physical security. Attackers who can physically access a BLE-based device will be able to pair with it and may control it thereafter. Therefore, manufacturers may implement extra authentication mechanisms at the application layer to address this issue. In this paper, we design and implement a BLE Security Scan (BLESS) framework to identify those BLE apps that do not implement encryption or authentication at the application layer. Taint analysis is used to track if BLE apps use nonces and cryptographic keys, which are critical to cryptographic protocols. We scan 1073 BLE apps and find that 93% of them are not secure. To mitigate this problem, we propose and implement an application-level defense with a low-cost $0.55 crypto co-processor using public key cryptography.**

***Index Terms*—Bluetooth Low Energy, IoT Security, BLE attacks, Reverse Engineering, BLE Security Scanning.**

I. Introduction

We have entered the age of Internet of Things (IoT), featuring various applications, such as smart healthcare, smart home, and smart city. As a promising wireless network technology, Bluetooth Low Energy (BLE) has positioned itself as one of the key enabling technologies in the IoT [1], [2]. Compared to the Bluetooth Classic, BLE maintains a larger coverage area while considerably reducing power consumption, which ideally meets requirements of IoT.

BLE achieves its communication security through a set of pairing strategies at the link layer, including *Just Works*, *Passkey-entry*, *Numeric Comparison* and *Out of Band* (*OOB*). A typical BLE application scenario is that the owner of two BLE devices pairs them together wirelessly through Bluetooth and wants to defeat the man-in-the-middle (MITM) attack. The Passkey-Entry and Numerical Comparison pairing protocols ensure the owner that he/she is pairing the two devices he/she sees and there are no MITM attacks.

However, the threat model of BLE pairing strategies is obsolete in the era of IoT. An implicit assumption for secure BLE pairing is that the owner owns the two pairing devices and the physical security of these devices is ensured. However, a BLE enabled IoT device may be deployed in a public environment. An attacker may access and pair with such a device. Therefore, a secure pairing strategy such as Passkey-entry and Numerical Comparison does not prevent the attacker from pairing with the victim device [3].

Given that the link layer BLE pairing strategies are obsolete [3], IoT vendors may resort to the application layer mechanisms for authentication. We carefully review the state-of-the-art of authentication protocols [4]–[7] and conclude a secure authentication protocol must involve cryptographic keys for encryption and nonces for message freshness. Without encryption, the communication is subject to eavesdropping. Without nonces, the communication is subject to replay attacks.

In this paper, we propose the **BLE** application **S**ecurity **S**canning tool (BLESS) to study the security practices of BLE products. BLESS scans a BLE app and checks if it uses cryptographic keys and nonces. If an app does not involve cryptographic keys or nonces, it is not considered secure. The challenge is how to identify if cryptographic keys or nonces are used by the BLE apps and the corresponding protocols. The intuition of BLESS is that data *flows*, and their sources and sinks can be used to identify the involvement of nonce and keys. For example, a key can be derived from the input of a user within an app. The app saves it onto its own disk for further usage and distributes it to a peer device. Nonces such as random numbers are often used in a challenge-response protocol with back and forth message exchanges. By utilizing these features, we use taint analysis [8], [9] to track data flows, sources and sinks. If an app does not have such features, the app does not use keys or nonces. We then determine that the app does not implement an application layer authentication protocol, and is not secure.

To validate our tool, we performed several case studies of BLE products, including popular blood pressure monitors from Smart Pulsewave and BP3L at Amazon. These two products are denoted as insecure by our tool. Specifically, without keys and nonces, blood pressure monitors from Smart Pulsewave are subject to eavesdropping and replay attacks. The BP3L blood pressure monitor uses a hard-coded key and is subject to spoofing attacks. Anybody who extracts the hard-coded key from the app can pretend to be the blood pressure monitor and bypass the authentication of the app. This allows us to forge the blood pressure reading data sent by the blood pressure monitor. Due to the fake blood pressure reading, a patient's life may be in danger for medication mistakes and wrong medical treatment accordingly.

The major contributions of this paper can be summarized as follows.

1) We design and implement BLESS. To the best of our knowledge, we are the first to use taint analysis to determine whether an app implements an application layer authentication protocol.
2) We apply BLESS to BLE apps, although it can be used in other contexts (e.g., Wi-Fi, Zigbee and so forth).We obtain 1073 BLE apps from androzoo [10] and find that 76% of the apps do not implement authentication protocols, and 93% are not secure.
3) The recent advance of hardware allows us to implement an application level authentication protocol with a low-cost $0.55 crypto coprocessor, ATECC608A, based on public key cryptography. We evaluate its performance and show its feasibility in IoT BLE applications.

The rest of the paper is organized as follows. We begin with a brief introduction to BLE in Section II. Section III presents the design and implementation of BLESS. In Section IV, we preform case studies to validate our design. Section V presents our countermeasures. Section VI evaluates the performance of BLESS and our countermeasures. We study related works in Section VII and conclude the paper in Section VIII.

II. Bluetooth Low Energy

This section provides a brief introduction to BLE. We first present how two BLE devices are connected. Afterwards, we introduce how the pairing process occurs between them. Finally, we show how data is organized and accessed by BLE devices through Attribute Protocol (ATT) and the Generic Attribute Profile (GATT).

A. Connection Setup

We take the connection setup procedure between a smartphone and a smart lock as an example. An app on the smartphone is designed to communicate with the smart lock. First, the smart lock broadcasts *advertising* packets which indicate that the smart lock is connectable. The app on the smartphone receives these *advertising* packets, and then sends a *scan request* to the smart lock to get more information from it. Afterwards, the lock responds with a *scan response* packet. The *advertising* packets and the *scan response* packet include the basic information of the smart lock, such as the device name, primary service description, and so forth. Based on the information above, the app on the smartphone can decide whether the smart lock is the device of interest. If so, the app on the smartphone sends the *connect request*, and the connection is established. According to BLE specifications, the device which sends the *connect request* is called *master device*, while its peered device is called *slave device*.

B. Pairing Process

In BLE, two parties can communicate in plaintext with each other. They may also go through the pairing process to negotiate keys and encrypt the communication at the link layer.

Pairing involves three phases. In the first phase, the two devices exchange their pairing features. Based on these features, a suitable pairing method can be adopted for the next phase. In the second phase, the two devices agree on a long term key (LTK) for future link encryption. In such a way, every time they want to communicate with each other, the pairing process will not be repeated. There are four pairing methods provided by BLE as of now, including *Just Works*, *Passkey Entry*, *Numeric Comparison* (Only for BLE version 4.2 and beyond) and *Out of Band*. Among them, *Just Works* is subject to MITM attacks [11], while others can defend against such an attack. Therefore, the generated key also has two security properties, namely authenticated-and-MITM-protection and unauthenticated-and-no-MITM-protection. In the third phase, vicesеAt08980(2916530637.1F97.ocol3063JT* A)110.10TT16.90637.1de]T an Identity Resolution Key (IRK) and Connection Signature Resolving Key (CSRK) are generated from one device (either the master or the 29979(maea2(wit4T*[()-4f16.908980(299ing)-o.08980(29 encrypted read/write, authenticated read/write and authorized read/write. The read/write attribute can be accessed freely, while encrypted read/write can be accessed only when the link is encrypted. The authenticated read/write attribute can only be accessed when the link is encrypted by a key with the property of authenticated-and-MITM-protection (when Passkey-Entry, Numeric Comparison or OOB is applied). However, how the authorized read/write attribute can be accessed is not specified in BLE specifications as of now.

Generic Attribute Profile (GATT) is built upon ATT. It organizes attributes into *services*. GATT allows the devices to exchange arbitrary data in the format of attributes. By use of GATT, two peer devices can interact in a Client-Server Architecture. The device holding many attributes with data is the server, while the other device requesting data from the server is the client. Services may include other services as their building blocks. The major service that contains subordinate services is called the primary service, while the auxiliary ones refer to the secondary services.

III. BLESS: BLE App Security Scanning

In this section, we first present the assumption of our security scanning procedure. We then introduce the scanning strategy to identify encryption and authentication in BLE apps. Afterwards, we introduce implementation details of the scanning strategy.

A. Assumption

We assume that an app does not rely on BLE pairing for security given BLE's vulnerabilities. Recall that an attacker can pair with a victim BLE device that is deployed in public. The attacker may also install malware on a victim's smartphone and deploy co-located attacks [12]–[14]. Therefore, a BLE app shall use BLE only as a wireless communication venue and implement security patterns at the application layer. The BLE app implements encryption for confidentiality and two types of cryptography based authentication services: integrity authentication for verifying data integrity and source authentication for verifying the user identity. For example, password based source authentication can be used to verify the user, while encryption can be applied to protect the transmission of the password via the BLE wireless channel. We assume that an attacker is not present during the bootstrapping process.

B. Pattern of Keys and Nonces

A secure authentication protocol must involve keys to ensure confidentiality and the integrity of data transmission. Nonces are needed to ensure the freshness of messages. If no key or nonce is used in an app, the app is not considered secure. When keys and nonces are used in cryptographic protocols, their use follows specific patterns of data flows. BLESS is designed to identify apps which utilize insecure data flow patterns.

We now introduce patterns of keys. In a cryptographic protocol implemented by BLE apps, a master key must be shared with both sides (e.g., a smartphone and a device) through BLE communication. The master key can be used to generate other keys such as session keys and will be saved onto storage for future use. A master key can be a user-defined key, smartphone-defined key, or device-defined key. Note that in all instances, both the smartphone and peer device store the master key for future use.

- A user-defined master key is generated from a user, stored on the smartphone, and shared with the peer device.
- A smartphone-defined master key is generated from a random number generator, stored on the smartphone, and shared with the peer device.
- A device-defined master key is generated from the peer device, stored on the device, and shared with the smartphone.

We now introduce patterns of nonces. In a cryptographic protocol, a nonce is usually generated by a device and a smartphone, and exchanged by both sides for the purpose of guaranteeing the freshness of messages and fighting against replay attacks. There are three type of nonces:

- Random number: A random number can be generated by a smartphone or its peer device. It is sent from one side to the other, and will go back to the original device as in the challenge-response protocol. It is not saved onto storage.
- Sequence number: A sequence number increments its value every time it is used. A limitation of this is that the sequence number will eventually reach its maximum value and wrap around. Therefore, for message freshness, a *rekeying* process will use a random number to generate a session key which secures the sequence number. Therefore, for a secure cryptographic protocol, sequence numbers must always involve an element of randomness.
- Time-stamp: A time stamp is generated from a date and time function. This value is guaranteed to always be unique when it is generated, as long as the source of the time stamp is reliable.

C. Our Solutions

Based on the patterns introduced in Section III-B, we can conduct taint analysis to check if keys or nonces are involved in an authentication protocol. Taint analysis can build a data flow from a specific entry point, known as the source, to a specific exit point, known as the sink. Particularly, we can identify a key or a nonce based on the sources and sinks of data flows. To this end, we taint the functions that may generate a key or a nonce as sources, and taint the BLE communication APIs and data storage APIs as sinks. We taint BLE communication APIs since authentication only occurs when the BLE app and its peer device exchange data. These BLE communication APIs will not change, even if heavy obfuscation is adopted. For example, the *writeCharacteristic(.)* function will write a byte array into the device, while *readCharacteristic(.)* will obtain data from the device. The callback function *onLeScan(.)* can collect information from the scan responses and advertisement packets. We taint data storage APIs because a key can be saved onto disk after generation.

Algorithm 1: Key/Nonce searching algorithm

```
Data: Paths ⟸ Taint paths get from the app
Result: KeySet, NonceSet ⟸ collected keys and nonce
        from the application
Initialization:
RandomSet = $\phi$; KeySet = $\phi$;
static APKInfo {
shareDeviceData=false; sharePhoneData=false;
  shareUserInput=false; saveUserInput=false;
  saveDeviceData=false; savePhoneData=false;
  Sequence=false; Timestamp=false;
} ;
while foreach path in Paths do
    analysisPath(path,APKInfo);
end
if APKInfo.sharePhoneData then
    if APKInfo.savePhoneData then
        KeySet.add($Key_{phone}$);
    else
        nonceSet.add($Rand_{phone}$)
    end
end
if APKInfo.shareUserInput then
    if APKInfo.saveUserInput then
        KeySet.add($Key_{user}$);
    end
end
if APKInfo.shareDeviceData then
    nonceSet.add($Rand_{device}$)
end
if APKInfo.saveDeviceData then
    KeySet.add($Key_{device}$);
end
if APKInfo.Sequence then
    nonceSet.add($Sequence$)
end
if APKInfo.Timestamp then
    nonceSet.add($Timestamp$)
end
return KeySet, nonceSet;
```

Algorithm 2: Path analysis algorithm

```
Data: path,APKInfo
Initialization:
source=path.get_BLE_related_Source();
sink=path.get_BLE_related_Sink();
if source.fromDevice() and sink.toDevice() then
    APKInfo.shareDeviceData=true;
end
if source.fromDevice() and sink.saveToDisk() then
    APKInfo.saveDeviceData=true;
end
if source.fromDisk() and sink.toDevice() then
    APKInfo.Sequence=true;
end
if source.readTime() and sink.toDevice() then
    APKInfo.Timestamp=true;
end
if source.fromPhone() and sink.saveToDisk() then
    APKInfo.savePhoneData=true;
end
if source.fromPhone() and sink.toDevice() then
    APKInfo.sharePhoneData=true;
end
if source.fromUser() and sink.toDevice() then
    APKInfo.shareUserInput=true;
end
if source.fromUser() and sink.savetoDisk() then
    APKInfo.saveUserInput=true;
end
```

A running example: To better understand our approach, we take the Ultraloq smart lock as a running example. The Ultraloq is a Bluetooth-enabled smart lock that enables a user to control the lock remotely on a smartphone. In its authentication protocol, a key and a random number are used to secure the communication. To generate a key, the owner of the lock is required to set a password. The password is hashed and saved to the smartphone's disk. The smartphone then shares the hashed password with the smart lock. The hashed password will be used as a master key in further communications. To ensure the freshness of messages, the lock also performs a challenge-response protocol each time the app sends a command. For example, when the user tries to open his door, the smart lock first sends the app a random number as a challenge. The app receives the challenge, and feeds the hashed password along, the control command, and the challenge into an encryption function to generate a response. Afterward, the app sends the response to the smart lock. The smart lock will unlock the door accordingly, when the random number and the pre-shared key are matched.

We first demonstrate how to determine if the app uses a key. We taint the sources and sinks respectively. We determine that there are two possible paths in total: (i) a path whose source is a function that can collect user inputs and whose sink is a function that can write data into a BLE device, which indicates that a value is generated from a user's input and shared with the peer device; (ii) a path whose source is a function that can collect user inputs and whose sink is a function that can save the value to the smartphone's storage, which indicates that a value is generated from a user's input and saved. If the above paths are identified, we can know that a key may be adopted in authentication. We then demonstrate how to determine if the app uses a random number in this case. We taint the sources and sinks respectively. In this case, the taint path has the following feature: The source of the path is a BLE data reading function, while the sink of the path is a BLE data writing function. This indicates that the device sends a value to the smartphone, and then the smartphone sends it back after some processing. This is a typical case of using a random number to perform the challenge-response.

The above example is one possible case which demonstrates our approach. We introduce algorithms that identify all the possible paths automatically and evaluate the security of apps. Algorithm 1 traverses all data paths and returns a set of identified keys and nonces in an application. $APKInfo$ is a set of Boolean values which stores specific behaviors of the app, based on the source and sink of a data flow. For instance, $APKInfo.shareDeviceData$ is set to *true* if the source and sink of a data path are identified as the peer device, since this indicates that a message (e.g., a random number) flows from the peer device, to the smartphone, back to the peer device. Algorithm 2 is invoked by Algorithm 1 to evaluate each taint path from sources to sinks in order to construct $APKInfo$. Moreover, to determine whether an app is secure, we introduce Algorithm 3. Specifically, after taint analysis, there are 4 cases in total: (i) An app that does not use either a key or a nonce is not secure. Its communication is subject to replay attacks and fails to ensure data integrity. (ii) An app not using a key fails to ensure the data integrity, and suffers from spoofing attacks and eavesdropping attacks. (iii) An app that does not use nonces will suffer from replay attacks. (iv) An app that uses keys and nonces is potentially secure from these attacks. However, security cannot be guaranteed, since an app may use the key or nonce in an insecure way.

Algorithm 3: Vulnerabilities Detection algorithm

```
  Data: Paths ⟸ Taint paths get from the app
  Result: VulSet ⟸ Vulnerabilities
if KeySet == $\phi$ and NonceSet == $\phi$ then
    VulSet.add($Vul_{replay}$);
    VulSet.add($Vul_{nointegrity}$);
else
    if KeySet $\neq \phi$ and NonceSet == $\phi$ then
        VulSet.add($Vul_{replay}$);
    else
        if KeySet == $\phi$ and NonceSet $\neq \phi$ then
            VulSet.add($Vul_{nointegrity}$);
        end
    end
end
return VulSet;
```

D. Implementation

BLESS extends the Amandroid framework [9], which provides prerequisites for taint analysis. We use this framework because it can handle Inter-Component Communication (ICC). ICC is a mechanism that allows different components (e.g., "activities") to communicate with each other. BLE apps often involve ICC. For example, a BLE app may obtain data from advertisement packets in scanning Activity and pass it to controlling Activity. The tool must be equipped to handle ICC, otherwise it will invoke false positives, where secure apps are mistakenly identified as insecure. Particularly, by customizing the profile "TaintSourcesAndSinks" in Amandroid, we are able to fully trace the taint path of interest. After identifying all these paths, we apply the algorithms in Section III-C to evaluate the security of an app.

Tainting Sources: A taint source can be either of the following cases: a source from a user, a source from a smartphone, and a source from a device. To identify a source from a user, the API *getEditText()* is tainted, which allows us to focus on the data from user inputs. To identify a source from a smartphone, (i) we taint functions that can generate a random value, such as *java.util.Random.nextBytes* and *java.util.Random.nextLong*; (ii) we taint functions that can get the current system time such as *System.currentTimeMillis()*; (iii) we taint functions that can read data from storage, including file reading functions like *FileReader.read()*, database operation functions such as *Cursor.getString()*, and profile operation functions such as *SharedPreferences.getString()*. Note that smartphone defined keys, random values, serial numbers and time-stamps may be generated from (i) and (ii). To identify a source from a device, (i) we taint functions that can read data from device services, such as *getValue()* and *gattCharacteristic*, which indicate that the source is from a device's GATT Services; (ii) we taint functions that can read data from advertisement packets and scan response packets, since these packets may offer random numbers. Different from framework APIs, the android system uses the callback *OnLeScan()* to handle the advertisement packets. The parents of these callback functions may be located in different packages and classes. To taint advertisement packets, we have implemented our own subclass that extends the class *AndroidSourceAndSinkManager* in their framework and overwrites the function *isCallbackSource()*. The framework can then filter these sources while performing the analysis.

Tainting Sinks : A taint sink can be either of the following: (i) a sink that relates to data storage on the smartphone, i.e., functions that can write data to a smartphone's storage, such as *FileWriter.write()* and *SQLiteDatabase.insert()*; (ii) a sink that relates to writing data onto a peer device, i.e., functions that can write out the data through BLE, such as *setValue()*.

IV. Security Analysis of BLE-enabled Applications

In this section, we perform security analysis of BLE-enabled applications to validate our tool. Two examples are presented in this section, including blood pressure monitors from BP3L and Smart Pluswave. Both devices are denoted as insecure since the Pluswave does not use a key o(y)-43(a)-non5(vi(o(y)35(ans)-2 app. Specifically, static analysis is used to identify the authentication protocol adopted by the application, while dynamic analysis is used to identify the authentication parameters (e.g. a key) used in authentication protocol.

1) Static analysis: An attacker may want to obtain the source code and understand the details of the authentication protocol. We employ *APKTool* [15] to obtain *Smali* code from an Android Package (APK) and use *Smali2Jar* to convert *Smali* code into the Java format. If this conversion fails, We read *Smali* code directly. Heavy obfuscation [16] may be employed to defend against reverse engineering of an app. Direct use of Android APK decompilers does not work against obfuscation. Therefore, we adopt source code instrumentation [17] to explore the workflow, inputs, temporary values and outputs of interest. Source code instrumentation also enables us to directly use some functions as the building blocks of our customized app for testing and attacking. When performing static analysis of an app, we mainly focus on the functions that communicate with the peer BLE device. Tracing these functions can reveal the core engine of the authentication protocol, such as how a command is generated, and how an app resolves an authentication message from the device.

2) Dynamic Analysis: Static analysis is not omnipotent since it cannot trace the outputs generated by different inputs. In other words, static analysis may not observe the impact of various parameters on the authentication process. Therefore, we resort to dynamic debugging to address this issue. We use the *hook* technology to trace each authentication parameter (e.g., a random number) of an app. Xposed [18] is a popular hook framework which can record, modify and replay the inputs and outputs of a function. We use Xposed to write our own code in order to observe the changes of the workflow, as well as the variation of the authentication parameters. For example, hooking the function *writeCharacteristic* enables us to observe how the write commands, which are sent by the app, change in different contexts. Hooking the function *readCharacteristic* allows us to obtain the value of an attribute set by the device.

B. Case Studies

With the analytic techniques above, we now present the discovered attacks that are able to compromise blood pressure monitors from Pluswave and BP3L. From these attacks, we can observe that our tool can detect the vulnerabilities of apps effectively.

1) Smart Pluswave: According to our analysis, the blood pressure monitor from Smart Pluswave does not use a key or a nonce to implement authentication. The confirm this, we first launch a spoofing attack by creating a fake device that acts as the original blood pressure monitor. Therefore, the smartphone app will send all its authentication data and control command to our fake device. For example, we discover that the byte array "cc80020301030003" is an encoded command that is used to start the measurement. When the user is off-line, we can launch a replay attack by re-sending the encoded control command and authentication data. In this way, we can take control of the blood pressure monitor.

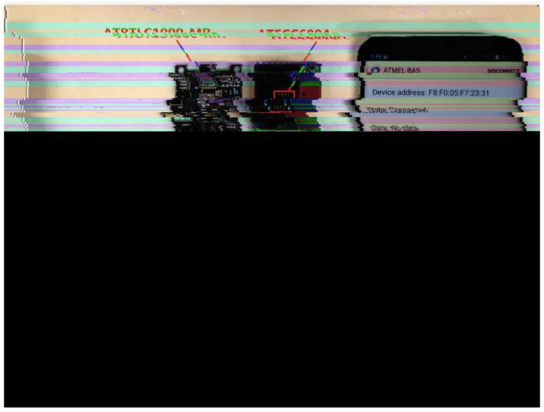
2) BP3L Blood Pressure Monitor: We now demonstrate how we can compromise the Blood Pressure Monitor from BP3L. According our experiment, the BP3L does not use a key to secure communication. We first present the workflow of the blood pressure monitor. At first, the device broadcasts the basic information, including the manufacturer data and device name. The smartphone receives the data and checks if the device belongs to the specific vendor. Authentication is then performed. Specifically, the device and the smartphone use the challenge-response protocol with a fixed key value to perform authentication. When the authentication is done, the application receives a fixed confirmation message from the device, indicating that the smartphone and blood pressure monitor are ready to communicate. The blood pressure monitor will encrypt a measurement and send the encrypted data to the smartphone. The smartphone will decrypt it with a function named *deciphering* and show the data to the user. The key used to encrypt and decrypt the data is generated from a function *getKey*, which uses the hardware version number of the device as its input. As long as the hardware version number does not change, then *getKey* will produce a fixed key.

We can deploy a spoofing attack on the blood pressure app without any changes. Specifically, we create a fake blood pressure monitor that acts as the original one. In the original authentication process, the monitor sends the smartphone a confirmation message to show that the monitor is authenticated. Since the confirmation message is fixed, our fake blood pressure monitor can send the message to the smartphone. In this way, we bypass the authentication process. Afterwards, we re-write an encryption function based on the source code, then feed the function the key and the fake blood pressure measurement to generate the cipher text. We send the generated cipher text to the smartphone to deploy the fake data injection attack.

It can be observed that there are two fixed keys in the case of BP3L. One key is used to perform the challenge-response protocol, while the other is used to encrypt the measurement from the blood pressure monitor. However, these two keys fail to ensure data integrity, since they can easily be extracted by an attacker. This case study indicates that it is not sufficient to only detect apps without keys. An app that uses a fixed key, which is not secure, should also fall into the same category. Moreover, an attacker can control a victim device as long as an official app is installed on his smartphone.

V. Countermeasure

As discussed earlier, we assume that an attacker is not present during the bootstrapping process. However, if an attacker can perform the sniffing process at the initiation of password setting, he can obtain the key and control the smart device freely. In this regard, we present an application level defense that enhances the security of BLE based apps.


ATBTLC1000-MR
ATECC508A
ATMEL-BAS
DISCONNECT
Device address: F8:F0:05:F7:23:31
State: Connected
Data: No data

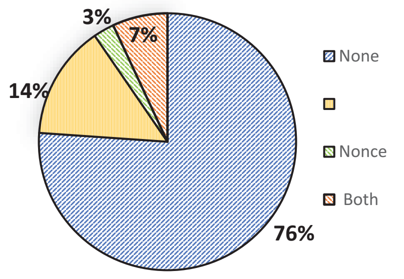


Fig. 3: Security State of BLE Products

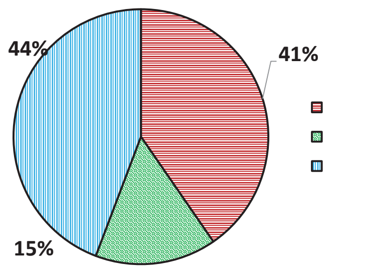


Fig. 4: Ratio of different types of keys that used by device

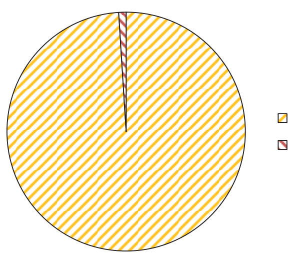


Fig. 5: Ratio of different types of nonces that used by device

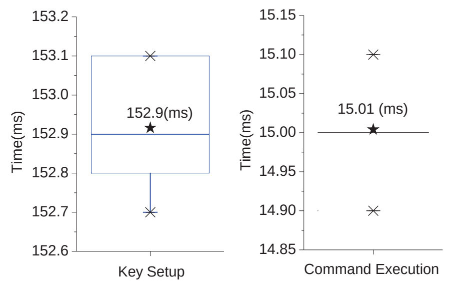


Fig. 6: Command Execute Time

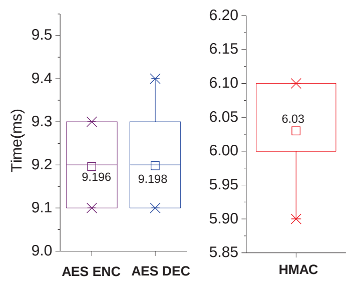


Fig. 7: Time for AES and HMAC

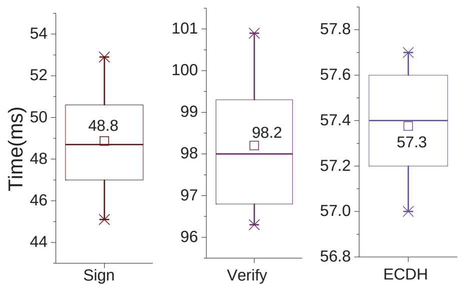


Fig. 8: Time for ECDSA/ECDH

TABLE I: Accuracy Comparison

Tool	Secure Apps	Insecure Apps	Throw Errors
BLESS	75 (86%)	100 (100%)	0 (0%)
BLECryptracer	7 (10.7%)	94 (94%)	6 (6%)

Figures 4 and 5 show the proportion of keys/nonces of different types used by BLE apps. We can observe that user-defined keys are more widely used than smartphone-defined keys or device-defined keys. The user-defined key is more feasible when compared with others. As long as the user remembers his/her password, he/she can always perform the authentication process without resetting a device or backing up configurations. We also find that developers prefer to let the device generate nonces, which will let the device verify commands sent from the smartphone. This is because the smartphone is a master device which controls its peer device. Replay attack usually occurs when the smartphone sends a command to its peer device; in a BLE application, the peer device does not typically send control commands to the smartphone.

B. BLESS vs. BLECryptracer

BLECryptracer is a BLE security detection tool introduced by Sivakumaran *et al.* [14]. The key insight of their tool is to identify the cryptographic API existing in BLE Apps. That is, a BLE app is considered secure if and only if cryptographic APIs are detected. In this section, we will perform a comparison between BLESS and BLECryptracer in terms of accuracy. There is no dataset of wild BLE apps that are considered as secure. Therefore, we must manually check the security of 75 apps that are denoted as secure by BLESS in section VI-A. The principles used for manual analysis is whether the app has a key and a nonce. 65 of these apps are secure according to our manual analysis, which means that BLESS identified 10 false negatives in its analysis. Then, we use BLECryptracer to perform a similar detection process. Table I shows the results. It can be observed that BLECryptracer can only detect 7 secure apps, since these secure apps contain Java Cryptography Architecture, such as the *java.security and* javax.crypto. Their tool will fail when an app implements a customized cryptographic algorithm. For example, the smart lock Ultraloq uses customized cryptographic algorithm and denoted as insecure by BLECryptracer.

Similarly, there is no dataset of wild BLE apps that are considered insecure. We manually checked 100 apps that are denoted as insecure by BLESS in Section VI-A as our testing dataset. It can be observed from Table I that 94 apps are denoted as insecure by the BLECryptracer. Furthermore, there are 6 apps that can not be analyzed by BLECryptracer. BLECryptracer throws errors when processing these apps. Although we can not see much difference from this experiment, we argue that BLESS has advanced features when compared with BLECryptracer: (i) As demonstrated in section V, our tool can handle an app that uses a fixed key, while BLECryptracer can not. Therefore, BLECryptracer may report a false alert when dealing with such apps. In their paper, the authors introduce another tool named CogniCrypt to identify this case. (ii) Their tool does not take nonces into consideration, which is not comprehensive. (iii) Their tool does not take the data exchanged via advertisement packets into consideration. This is also not comprehensive, since an app may receive keys or nonces from advertisement packets. An app named "com.flyjiang.dongha.activity" uses advertisement packets to receive a random number from its peer device.

C. Application Layer Defense with ATECC608A

1) Command execution time: Fig. 6 shows the time for executing a command on Microchip's Atmel Samd21 development board with ARM's Cortex-M0+ MCU operating at 32 MHz. We evaluate two metrics: (i) the time for decrypting a received command with integrity checking; (ii) the time for performing the authenticated ECDH with ECDSA. We run each test 50 times. It can be observed that the average runtime for decrypting a received command and performing the authenticated ECDH is 15 ms and 150 ms, respectively.

2) Crypographic operation performance: Figs. 7 and 8 show the cryptographic operation performance of ATECC608A on Samd21. Fig. 7 gives the time for AES encryption/decryption and HMAC. Fig. 8 gives the time for performing ECDH/ECDSA. The input of AES and HMAC is 1 block (16 bytes), and the input of ECDSA is 2 blocks (32 bytes). We run each operation 50 times. The average time of AES encryption and decryption on ATECC608 is around 9 ms. The average time of performing HMAC is around 6 ms. The average time for performing ECDSA alone by the device is around 100 ms. The average time measured at the device side for performing ECDH alone without ECDSA is around 57 ms. The data is consistent with the performance data for the authenticated ECDH with ECDSA together. Fig. 8 also provides the average time for verifying the ECDSA signature at the device, which is around 50 ms. Note that the device does not need to perform this ECDSA signature verification in our context, although it can be used in other applications.

VII. Related Work

In this section, we review the most relevant works. From the discussion in previous sections, it can be observed that our work in this paper is quite different from other related works. Our work presents a novel security scanning tool for BLE enabled applications. In addition, we include relevant attack examples and hardware based countermeasures.

We first present Bluetooth sniffing tools and focus on the open source tools. FTS4BT Bluetooth protocol analyzer and packet sniffer is a commercial tool and often costs tens of thousands of dollars [26]. Michael Ossmann presented Ubertooth One, the first open-source low-cost Bluetooth test tool, at Shmoocon 2011 [27]. In 2013, Mike Ryan built a BLE sniffer on Ubertooth and demonstrated that the passkey pairing method for LE legacy connections was not secure. He developed a tool, *crackle*, which is able to crack such connections [28]. The Adafruit Bluefruit LE sniffer was introduced in 2014 [29]. BlueEar was built upon Ubertooth in 2016 [30] and able to sniff the traffic of Bluetooth Classic.

We now review recent survey papers related to BLE security. Hui Jun Tay et al. presented a survey of the vulnerabilities in the BLE beacons [31] and provided an overview of the current state of iBeacon security by summarizing three vulnerabilities (spoofing, DOS, and Hijack) in beacon platforms, and citing specific case studies. Hassan et al. summarized major security threats in Bluetooth Classic and BLE communication and discussed mitigation techniques [32]. Celebucki et al. [33] presented security features and shortcomings of BLE, Zigbee, and Z-Wave protocol. As for BLE, they pointed out that devices utilizing the legacy mode pairing were vulnerable to MITM attacks *during the pairing process*.

We now review related work on specific BLE attacks. Ryan et al. [28] showed a method that can brute fore the encryption of the BLE link layer. Jasek et al. [34] discovered a set of attacks between an mobile app and its peer devices. Their attack vectors include replay attack and brute force attack. Zegeye et al. cracked the BLE temporary key used in the pairing process by using the brute-force attack [35], which extended the attack in [28]. All these works do not involve a framework to detect the proposed attacks. Pallavi Sivakumaran et al [14] presented the Co-located attack, through which a malware can access the sensitive data on its peer device. In their work, they also proposed a detection framework. However, their detection framework determined whether an app is secure by testing the involvement of cryptographic operations. As discussed earlier, this may cause false alerts.

We now review related work on BLE security enhancement. Muhammad Naveed et al. [12] developed an OS-level protection mechanism to identify the binding relationship between an app and a device, and then used the relationship as the security policy. Giwon Kwon et al. [36] proposed a security method that can increase the length of the temporary key (TK) in BLE pairing, which could thwart the TK brute-force attack presented by Mike Ryan. Thrinatha [37] presented a countermeasure of MITM attacks by applying the anti-jamming techniques to the SSP model.

VIII. Conclusion

In the era of IoT, a BLE app should implement authentication protocols at the application layer since BLE devices may be deployed in public, and anybody may pair with it and use it. In this paper, we design a BLE security scanning tool (BLESS) to examine the security and identify the vulnerabilities of Android BLE apps. We use taint analysis to identify keys and nonces by exploring their data flow patterns. We find that at least 93% of these apps are not secure. To defend against those attacks, we propose and implement a practical application layer defense protocol with a low-cost ($0.55) crypto co-processor ATECC608A for authentication and authorization. Extensive evaluation is performed to validate the application level defense measure.

Acknowledgements

Jian Weng was partially supported by National Key R&D Plan of China (Grant Nos. 2017YFB0802203, 2018YFB1003701), National Natural Science Foundation of China (Grant Nos. U1736203, 61825203, 61732021), Guangdong Provincial Special Funds for Applied Technology Research and Development and Transformation of Important Scientific and Technological Achieve (Grant Nos. 2016B010124009 and 2017B010124002). Yue Zhang was partially supported by National Natural Science Foundation of China (Grant Nos. 61877029,61802145, 61872153,61872153,11871248, U1636209,61702222). Zhen ling was partially supported by National Key R&D Program of China 2018YFB2100300, National Science Foundation of China under Grant 61972088, 61532013, and Jiangsu Provincial Natural Science Foundation for Excellent Young Scholars under Grant BK20190060, in part by Jiangsu Provincial Key Laboratory of Network and Information Security under Grant BM2003201, in part by Key Laboratory of Computer Network and Information Integration of Ministry of Education of China under Grant 93K-9, in part by Collaborative Innovation Center of Novel Software Technology and Industrialization, and in part by Collaborative Innovation Center for wireless Communications Technology. This work was also partially supported by US National Science Foundation under Grant Nos. 1931871, 1915780 and 1642124. Any opinions, findings, conclusions, and recommendations in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] The Wikipedia, "Bluetooth low energy," https://en.wikipedia.org/wiki/Bluetooth_Low_Energy.

[2] Y. Zhang, J. Weng, R. Dey, and X. Fu, *Bluetooth Low Energy (BLE) Security and Privacy*. Cham: Springer International Publishing, 2019, pp. 1–12. [Online]. Available: https://doi.org/10.1007/978-3-319-32903-1_298-1

[3] Y. Zhang, J. Weng, R. Dey, Y. Jin, Z. Lin, and X. Fu, "On the (in)security of bluetooth low energy one-way secure connections only mode," 2019.

[4] M. Bellare and P. Rogaway, "Entity authentication and key distribution," in *Annual international cryptology conference*. Springer, 1993, pp. 232–249.

[5] M. Burrows, M. Abadi, and R. M. Needham, "A logic of authentication," *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, vol. 426, no. 1871, pp. 233–271, 1989.

[6] L. Lamport, "Password authentication with insecure communication," *Communications of the ACM*, vol. 24, no. 11, pp. 770–772, 1981.

[7] E. Barker and A. Roginsky, "Recommendation for cryptographic key generation," *NIST Special Publication*, vol. 800, p. 133, 2012.

[8] Z. Yang and M. Yang, "Leakminer: Detect information leakage on android with static taint analysis," in *2012 Third World Congress on Software Engineering*. IEEE, 2012, pp. 101–104.

[9] F. Wei, S. Roy, X. Ou *et al.*, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 1329–1341.

[10] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*. IEEE, 2016, pp. 468–471.

[11] Ronyip, "Mitm attack on "just works" pairing," https://www.silabs.com/community/wireless/bluetooth/forum.topic.html/mitm_attack_on_just-OoG9.

[12] M. Naveed, X. Zhou, S. Demetriou, X. Wang, and C. A. Gunter, "Inside job: Understanding and mitigating the threat of external device misbinding on android," in *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014.

[13] 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'19), San Diego, CA*, 2019.

[14] P. Sivakumaran and J. Blasco, "Attacks against ble devices by co-located mobile applications," *arXiv preprint arXiv:1808.03778*, 2018.

[15] Connor Tumbleson - Current Maintainer,Ryszard Wisniewski - Original Creator, "Apktool," https://ibotpeaches.github.io/Apktool/.

[16] Y. Zhang, J. Weng, J. Weng, L. Hou, A. Yang, M. Li, Y. Xiang, and R. Deng, "Looking back! using early versions of android apps as attack vectors," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–1, 2019. [Online]. Available: 10.1109/TDSC.2019.2914202

[17] A. Eustace and A. Srivastava, "Atom: A flexible interface for building high performance program analysis tools," in *Proceedings of the USENIX 1995 Technical Conference Proceedings*. USENIX Association, 1995, pp. 25–25.

[18] Simple Themes, "Xposed," http://repo.xposed.info/.

[19] N. Saxena, J.-E. Ekberg, K. Kostiainen, and N. Asokan, "Secure device pairing based on a visual channel," in *Security and Privacy, 2006 IEEE Symposium on*. IEEE, 2006, pp. 6–pp.

[20] M. INS, "Atecc608a," https://www.microchip.com/wwwproducts/en/ATECC608A.

[21] D. Johnson, A. Menezes, and S. Vanstone, "The elliptic curve digital signature algorithm (ecdsa)," *International journal of information security*, vol. 1, no. 1, pp. 36–63, 2001.

[22] Claire Swedberg, "Tamper-evident tag transmits whether or not seal breaks," https://www.rfidjournal.com/articles/view?16128=.

[23] Microchip, "Atmel crypto evaluation studio," https://www.microchip.com/developmenttools/ProductDetails/atmel/%20crypto%20%20studio%20(aces).

[24] M. Rouse, "Restful api," https://searchmicroservices.techtarget.com/definition/RESTful-API.

[25] i SENS, "Blood glucose meter," http://www.medicalexpo.com/product-manufacturer/i-sens-blood-glucose-meter-607-278.html.

[26] "Fts4bt$^{\text{TM}}$