BLESA: Spoofing Attacks against Reconnections in Bluetooth Low Energy

Jianliang Wu
Purdue University
wu1220@purdue.edu

Yuhong Nan
Purdue University
nan1@purdue.edu

Vireshwar Kumar
Purdue University
viresh@purdue.edu

Dave (Jing) Tian
Purdue University
daveti@purdue.edu

Antonio Bianchi
Purdue University
antoniob@purdue.edu

Mathias Payer
EPFL
mathias.payer@nebelwelt.net

Dongyan Xu
Purdue University
dxu@cs.purdue.edu

Abstract

The Bluetooth Low Energy (BLE) protocol is ubiquitously utilized to facilitate energy-efficient wireless communication among resource-constrained devices. To ease its adoption, BLE requires limited or none user interaction to establish a connection between two devices. Unfortunately, this simplicity is the root cause of several security issues.

In this paper, we studied, in particular, the security of the BLE link-layer focusing on the scenario in which two previously-connected devices reconnect. Our study started with a formal analysis of the reconnection procedure defined by the BLE specification. This analysis highlighted two critical security weaknesses in the specification. As a result, even a device implementing the BLE protocol correctly may be vulnerable to spoofing attacks.

To demonstrate these design weaknesses, and further study their security implications, we developed BLE Spoofing Attacks (BLESA) which enable an attacker to impersonate a BLE device and to provide spoofed data to another previously-paired device. BLESA can be easily carried out against some implementations of the BLE protocol, such as the one used in Linux. For BLE stack implementations in Android and iOS, we found another logic bug enabling BLESA. We reported this security issue to the affected parties (Google and Apple), and they acknowledged our findings.

1 Introduction

Bluetooth Low Energy (BLE) is the most widely utilized low-energy communication protocol, and by 2023, the number of devices using this protocol is expected to reach 5 billion [9]. The BLE protocol enables wireless, short-range communication which allows two devices to connect and exchange data. A typical usage scenario is having a smartphone using BLE to connect to a "gadget" device (such as a fitness tracker). Every BLE connection involves a device acting as a *client* (in this example, the smartphone) and another device acting as a *server* (in this example, the fitness tracker). The first time a user wants to connect the two devices together (allowing them to exchange data), a specific *pairing* procedure is performed, which varies depending on the type of the connected devices and their user-interfaces' capabilities.

The major factors that have helped the rapid growth of the adoption of BLE-enabled devices are its low cost and the bare minimal setup effort required for end users. Unfortunately, previous research [6, 11, 21, 23, 26] has shown that these user-friendly features have a negative impact on the security of this protocol. This concern is particularly worrisome since BLE communication is often used in security-sensitive devices, such as physical safety devices (e.g., locks) or health monitoring devices (e.g., medical implants) [15].

Researchers have pointed out and studied many implementation weaknesses in the BLE protocol by manually analyzing its specification [2, 7, 31, 34]. Additionally, some previous works have also performed a formal automated analysis of the BLE specification [4, 28]. However, these formal approaches only focused on limited aspects of the entire protocol, such as the BLE pairing mechanism. This limitation is due to the fact that it is challenging to fully formalize and automatically analyze the BLE specification. The challenges stem from the complexity of the BLE protocol, difficulties in modeling its multi-layer design, and its multiple variants and optional features (which allow this protocol to support a wide range of devices with significantly different capabilities).

In this paper, we study a previously-unexplored mechanism of the BLE protocol. Specifically, we focus on the mechanism of dealing with *reconnecting* two previously connected devices. This mechanism comes into play when, for instance, a server device (e.g., a fitness tracker) moves out of the BLE wireless communication range of a connected client device (e.g., a smartphone), and then at a later time, the two devices get close enough to be able to re-establish a connection.

To start our analysis of this scenario, we first carry out a formal verification of the relevant BLE link-layer specification [10], by using the ProVerif protocol verifier [8]. Our formal analysis highlights two weaknesses in the BLE official

specification. These weaknesses, in some BLE stack implementations, allow an attacker to launch a spoofing attack in which the attacker pretends to be a previously-paired server device and tricks a client device into accepting spoofed data.

The fundamental cause of these weaknesses is the fact that the specification allows implementing several aspects of the BLE protocol in multiple ways, some of which are vulnerable. For this reason, even BLE stack implementations correctly following the specification can potentially be susceptible to spoofing attacks. For instance, we found that the BLE protocol stack used in Linux client devices (e.g., Linux laptops), while following the BLE specification correctly, is susceptible to the identified spoofing attack.

Furthermore, we discover that even BLE protocol stacks implemented in ways that are, in theory, not susceptible to the identified attack are still vulnerable in practice due to a specific logic vulnerability. In particular, we found a similar implementation issue in the BLE stack used by Android devices and in that used by iOS devices. This issue makes many Android and iOS devices vulnerable to the identified attack. We have responsibly reported our findings to Google and Apple, and they have confirmed these vulnerabilities.

To showcase the identified issues, we design a novel practical attack, named BLESA (BLE Spoofing Attack). In BLESA, the attacker pretends to be a previously-paired server device, rejects the authentication requests coming from the client device, and then feeds spoofed data to it. We show that a Linux laptop running Ubuntu with the most updated version of its BLE stack is vulnerable to BLESA. In addition, we demonstrate the effectiveness of BLESA by launching it against Google Pixel XL phone recording the information from a wearable activity-tracking device called Oura Ring [24].

In summary, these are the main contributions of our work:

- We performed a formal analysis of the BLE link-layer authentication mechanism and discovered two design weaknesses in the BLE protocol specification.
- We studied how these design weaknesses affect existing BLE stack implementations, and we discovered a related logic bug in the BLE link-layer implementations used by Android and iOS devices.
- Exploiting the identified design weaknesses and the discovered logic bug, we crafted a novel attack, named BLESA, which allows an attacker to impersonate a server device and provide forged data to a previously-paired client device. We demonstrated the impact of BLESA by using it against a real-world activity tracker device and the Linux BLE protocol stack.
- We proposed mitigation techniques to the identified security issues, which enable systemic changes in the reconnection procedure to make it robust against BLESA.

2 Background

The original Bluetooth (Classic) protocol was designed for short-range wireless streaming applications such as audio, video, and file transfer. Different from Bluetooth Classic, Bluetooth Low Energy (BLE) is designed for energy-constrained low-cost IoT applications that require an intermittent transfer of smaller amounts of data at lower speeds.

The BLE protocol enables a device, acting as a server (e.g., a fitness tracker), to efficiently communicate relevant data (e.g., the number of tracked footsteps) to another connected device, acting as a client (e.g., a smartphone). In BLE, the typical communication procedure between the server and the client involves five steps:

1. The server advertises its presence by broadcasting advertising packets containing its identity.
2. The client scans for advertising packets to discover the server and establishes a wireless connection with it.
3. (Optionally) The client *pairs* with the server to share a long-term cryptographic key. This key is used to encrypt and authenticate the exchanged data.
4. The client initiates the request to access the data stored by the server.
5. The server checks if the client has the right to access the specific requested data. If this condition is true, the server grants read/write access to the requested data.

Security Levels. The data stored at a server is organized in *attributes* where each attribute (e.g., a blood pressure measurement) is identified using a unique identifier. BLE allows the server to protect attributes at a fine-grained granularity. In particular, the server can specify an access control policy for each attribute. Each such policy describes how an attribute can be accessed (read-only, write-only, or read-and-write), and which security level is needed to access it. Specifically, there are four possible security levels: no security (level 1), encryption only (level 2), encryption and authentication (level 3), and strong encryption with authentication (level 4).

When an access request is received from the client, the server checks whether the current connection meets the security level required to access the requested attribute. If the connection satisfies the required security level, the server grants access to the attribute. Otherwise, the server denies the request and sends an error message back to the client.

Pairing and Key Generation. As mentioned earlier, the pairing between the client and the server is optional, and it depends on the security level requested by the server. When encryption/authentication is needed to access an attribute, the client and the server must be paired to establish a (long-term) shared secret key. Once the long-term key is generated between two devices, the two devices are said to be *bonded*, and the long-term key is utilized to derive session keys to encrypt and authenticate the exchanged data during reconnections.

Different pairing methods require different levels of user interaction, and a specific pairing method is selected based on the available user interface (e.g., keyboard and display) of the pairing devices. For example, if the server device has a graphical user interface (such as in a smartwatch), user interaction (e.g., pressing a button or inserting a PIN) is required to approve the pairing on the server. In this case, the generated key can be used to access security level 3 or 4 attributes (depending on the length of the key). On the contrary, if the server device does not have input or output interfaces (e.g., a temperature sensor), no user interaction is possible on the server-side. In this case, the generated key can only be used to access attributes requiring security level 2.

It is important to notice that authentication and encryption are not always needed since data transmission in plaintext is allowed in the specification. In fact, the access control policy specified by the server fully determines if authentication and/or encryption are used. For example, the Polar H7 Heart Rate Sensor [30] transmits the heart rate in plaintext. Hence, no pairing is needed to read the heart rate from this device. Conversely, the Fitbit smartwatch [19] requires link-layer encryption and authentication. In this case, a smartphone has to pair with the smartwatch to be able to read its data.

3 Threat Model

We assume the adversary has the same capabilities as the Dolev–Yao model [18], i.e., the adversary can eavesdrop, intercept, and modify legitimate messages communicated between a server and a client. The adversary can also inject any message into the communication channel. However, the adversary does not know the secret key shared between the server and the client, and the cryptographic functionalities are perfectly secure. Also, the adversary cannot directly corrupt the firmware of the server and client. Since BLE is a short-range communication protocol, we assume the distance between the adversary and the client, and that between the adversary and the server are both within the Bluetooth range. In this scenario, the adversary aims to impersonate the server and to mislead the client with spoofed messages.

4 Formal Analysis of BLE Reconnection

We perform formal verification of the BLE link-layer authentication mechanism when the client reconnects with a previously connected server. We utilize ProVerif [8] to carry out this protocol verification.

4.1 ProVerif Model

We build a formal model for the authentication process when the client reconnects with the server in a new session. The authentication process is modeled as two communicating state machines, one for the client and one for the server. We model the state machine corresponding to the transmission of the attribute request of the client and the state machine corresponding to the handling of the attribute request on the server in the BLE specification [10]. The model comprehensively covers the different types of messages including those corresponding to the attribute access requests, attribute access responses, and the error responses. The communication between the client and the server is modeled as the free (or open) channel (`free plain_channel: channel`) where the adversary has the abilities as described in the threat model.

```
1  free attAccessReq [private].
2  free attAccessRsp [private].
3  free attSecAccessReq [private].
4  free attSecAccessRsp [private].
5  // Secrecy and integrity of attribute read request and
       attribute value
6  query attacker(attAccessReq).
7  query attacker(attAccessRsp).
8  query attacker(attSecAccessReq).
9  query attacker(attSecAccessRsp).
10 // Authenticity of attribute request and response
11 query event(clientRecvRsp) ==> inj-event(serverSendRsp).
12 // Consistency of request and response security levels
13 query x:SecLv; event(clientReqSecLv(x))==> inj-event(
       serverRspSecLv(x)).
```

Listing 1: ProVerif code with the verified security properties.

We also consider the optional features of the attributes to cover the different usage scenarios of server devices. As such, we model the attributes on the server as both readable and writable. We model two types of attributes: (1) *basic attribute* that can be accessed without pairing (i.e., at security level 1, no security), and (2) *sensitive attribute* that can be accessed after pairing (at security level 2 or higher). We model the client to send access requests to the server to read/write these two types of attributes in different orders. The shared secret key during the pairing is used to encrypt/decrypt the traffic when encryption is needed.

Security Goals. We analyze the above model in the light of traditional security goals that are presented in Listing 1. These security goals include: (1) *Confidentiality*, i.e., the communicated messages should not leak any sensitive data to the adversary (Line 6-9), (2) *Integrity*, i.e., the communicated messages should not be tampered by the adversary without being detected (Line 6-9), and (3) *Authenticity*, i.e., the communicated messages can be verified to be generated by the genuine sender (Line 11). Besides, we also check the BLE specific security goal extracted from the rules specified in the specification [10] when initiating requests from the client and handling requests on the server. For instance, only if the security level of the connection is consistent with the requirement to access the attribute, should the server give read/write access to the client (Line 13).

4.2 Discovered Design Weaknesses

Through formal verification, we discover several instances of violations of the checked security goals when the previously-connected devices reconnect. These violations result in the following two security weaknesses that can be potentially exploited by an adversary to launch spoofing attacks.

4.2.1 Weakness-1: Optional Authentication

We find that the link-layer encryption/authentication is optional, and the client and server may choose to disable it for a specific attribute. As such, in the case of the basic attribute, the confidentiality, integrity, and authenticity goals of the attribute access request and response can be violated as shown in the following results from ProVerif.

```
RESULT not attacker(attAccessRsp[]) is false.
```

Counterexample:

```
out(plain_channel, ~M) with ~M = attAccessReq at 2 in
copy a
in(plain_channel, ~M) with ~M = attAccessReq at 11 in
copy a_1
out(plain_channel, ~M_1) with ~M_1 = errCode at 13 in
copy a_1
in(plain_channel, a_2) at 14 in copy a_1
out(plain_channel, ~M_2) with ~M_2 = attAccessRsp at 16
in copy a_1
The attacker has the message ~M_2 = attAccessRsp.
A trace has been found.
```

We note that the server determines the security level of its connection with the client based on the requested attribute's access control policy. As the counterexample shows, when the access control policy allows the lowest security level (i.e., security level 1) of the connection, the attribute access request and the response can be transmitted in plaintext. In this case, no link-layer authentication is deployed, and therefore, the attacker can launch spoofing attacks against the server and client. We highlight that because the security level of the connection is guided by the server, the client cannot enforce a connection with higher security levels if the server only allows connections with the security level 1. This makes both the server and client are vulnerable to spoofing attacks.

4.2.2 Weakness-2: Circumventing Authentication

The BLE specification provides two authentication procedures when the client reconnects with the server after pairing.

(1) Reactive Authentication. In this procedure, the client sends the attribute access request in plaintext (i.e., at security level 1) right after establishing the connection. Only if the server responds with an error message revealing the inconsistency in the current security level of the connection and the one required for accessing the attribute, will the client react by enabling the encryption/authentication. In this case, the formal analysis indicates that the authenticity of the access response can be violated as shown in the following results.

```
RESULT inj-event(clientRecvRsp) ==>
inj-event(serverSendRsp) is false.
```

Counterexample:

```
event clientReqSecLv(secLv1) at 3 in copy a
out(plain_channel, ~M) with ~M = attAccessReq at 4 in
copy a
in(plain_channel, a_1) at 5 in copy a
event clientRecvRsp at 14 in copy a (goal)
The event clientRecvRsp is executed at 14 in copy a.
A trace has been found.
```

As the counterexample shows, the attacker can first intercept the request sent from the client. After that, the attacker can respond to the client with any message other than the error message, so that the client will receive and accept the message from the attacker. Although the sensitive attribute stored at the server is not leaked to the adversary, the adversary can eavesdrop on the request sent by the client, impersonate the server, and trick the client with a spoofed response corresponding to the sensitive attribute. We will demonstrate in Section 5.1 that the existing BLE stack implementation on Linux follows this reactive authentication making the corresponding clients vulnerable to spoofing attacks.

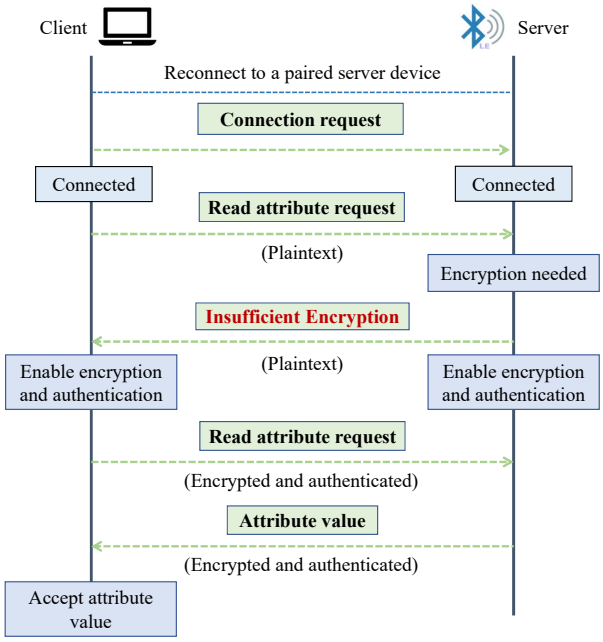
(2) Proactive Authentication. In this procedure, the client proactively enables the encryption/authentication before sending any request to the server. The client enables the encryption with the pre-shared secret key to proceed with the authentication. In this case, if the server fails to enable the encryption (which also means that it fails to enable the authentication), the client aborts the connection. Our formal verification shows that all the checked security goals hold during proactive authentication as shown by the following results.

```
RESULT not attacker(attSecAccessReq[]) is true.
RESULT not attacker(attSecAccessReq[]) is true.
RESULT inj-event(clientRecvRsp) ==>
inj-event(serverSendRsp) is true.
RESULT inj-event(clientReqSecLv(x)) ==>
inj-event(serverRspSecLv(x)) is true.
```

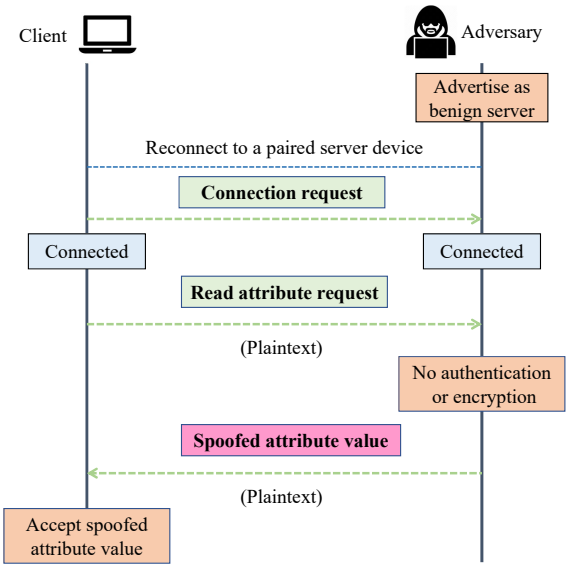
Unfortunately, as we will further elaborate in Section 5.2, the existing BLE stack implementations on Andriod and iOS fail to correctly follow this procedure making the corresponding clients vulnerable to spoofing attacks.

5 BLE Spoofing Attack (BLESA)

We exploit the design weaknesses identified through the formal verification to craft the BLE Spoofing Attack (BLESA) in which an adversary provides spoofed data to a client device pretending to be a previously-paired server device.



(a) Reconnection with the paired benign server.



(b) Instantiation of BLESA against the client.

Figure 1: BLE communication workflow when the client (following the *reactive* authentication procedure) reconnects with (a) the benign server and (b) the adversary which exploits the security weaknesses to circumvent the authentication procedure.

Attack Setup. We examine the scenario where the server and the client are securely paired in a previous session. Currently, they are disconnected but intend to start a new session, e.g., when the client moves out of the communication range of the server and then gets back. In this scenario, the adversary first discovers the server and connects with it to obtain the information (e.g., identifier) about the server's attributes. The adversary can easily obtain this information because the BLE protocol is designed to allow any device to connect with a BLE device and obtain the information about the attributes provided by it. Further, since the BLE advertising packets are always transmitted in plaintext, the adversary can easily impersonate the benign server by using its identity and cloning its MAC address. Then the adversary starts broadcasting spoofed advertising packets. This ensures that whenever the client attempts to start a new session with the previously-paired server, it can discover the spoofed advertising packets and establish a connection with the adversary.

Now the adversary is ready to launch BLESA against the client. Below we present the workflows of BLESA against the client following the reactive authentication procedure and that following the proactive authentication procedure.

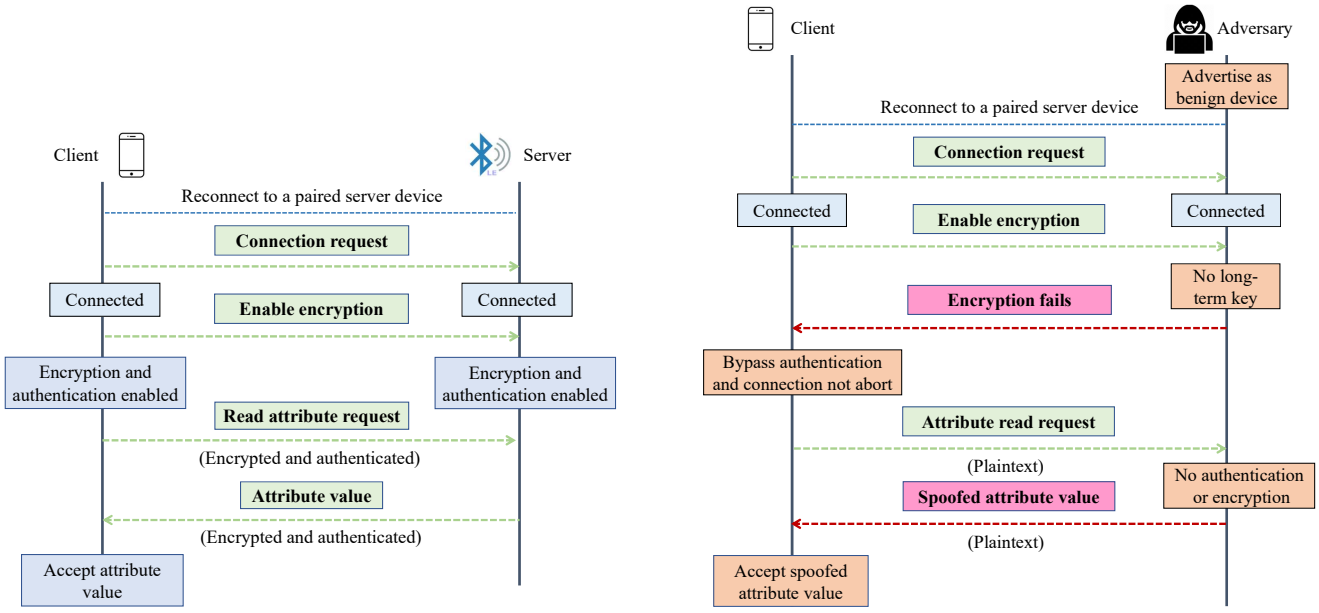
5.1 BLESA against Reactive Authentication

Figure 1a shows an illustration where the client and the server reconnect, and the reactive authentication procedure (discussed in Section 4.2.2) plays out in the benign environment. The client first sends an attribute read request to the server at the lowest security level without any encryption/authentication. If the attribute is sensitive and can be read only at a higher security level (e.g., security level 3 with encryption and authentication), the server would respond to the client with an error message (e.g., insufficient encryption). After receiving the error message, the client elevates the security level by enabling encryption and authentication using the pre-shared secret key and sends the request again. At this time, the server readily accepts the read request and returns the attribute value.

Now we present the workflow of BLESA in Figure 1b. Here, the adversary intercepts the attribute read request from the client and responds with the spoofed attribute value. Since the client does not encounter any error message, it mistakenly assumes that the attribute can be accessed at the lowest security level (i.e., in plaintext). As such, the client does not enable the encryption/authentication and therefore accepts the spoofed attribute value. We highlight that in this instantiation of BLESA, the critical dependency on the error message to adjust the security level of the connection makes BLESA successful against clients (e.g., the Linux laptops) which employ the reactive authentication procedure during reconnection.

5.2 BLESA against Proactive Authentication

Figure 2a shows an illustration of the workflow of the proactive authentication procedure when the client reconnects with the benign server. Right after the connection, the client re-



(a) Reconnection with the paired benign server.

(b) Instantiation of BLESA against the client.

Figure 2: BLE communication workflow when the client (following the *proactive* authentication procedure) reconnects with (a) the benign server and (b) the adversary which exploits the security weaknesses to circumvent the authentication procedure.

quests to enable the encryption (and authentication) using the pre-shared key, and the server complies with the request. Then the client securely sends an attribute read request, and the server securely responds with the (encrypted and authenticated) attribute value. If the BLE stack correctly implementing the proactive authentication follows the specification (as presented in Figure 10.2 on page 1382 in [10]), BLESA will fail against the client. However, our further analysis reveals that both Android and iOS-based client devices suffer from a logic bug that makes BLESA possible against these devices.

> ... If encryption fails either the bond no longer exists on the remote device, or the wrong device has been connected. The local device **must, after user interaction to confirm the remote device, re-bond, perform service discovery and re-configure the remote device**...

Figure 3: Description of the BLE protocol (page 1381 in [10]) indicating the steps to deal with the failure in enabling the encryption during the proactive authentication procedure.

Implementation Vulnerability. According to the BLE specification, when the client reconnects with the previously-paired server, if the encryption enabling process fails during the proactive authentication procedure, the client should either re-pair with the server (as mentioned in Figure 3) or abort the connection. However, the BLE stacks in some clients (i.e., Android and iOS-based devices) fail to follow the specification correctly. Specifically, we discover that even if the enabling encryption process fails, the client may not abort the connection and continue the communication in plaintext without re-pairing with the server. This flaw can be exploited by an adversary to launch BLESA.

> ... if the Host indicates that **a key is not available**, the slave shall either send an LL_REJECT_IND ... or **Key Missing** ... The Link Layer can now send LL Data PDUs and LL Control PDUs; **these packets will not be encrypted**.

Figure 4: Description of the BLE protocol (page 3031 in [10]) indicating the procedure to deal with the non-availability of the long-term key at the client device.

We suspect that this implementation vulnerability is very likely caused by the contradicted description across the BLE specification, as we elaborate below. Since encryption is not mandatory in BLE for accessing basic attributes and some server devices do not support link-layer encryption (i.e., the design weakness-1 discussed in Section 4.2.1), the BLE specification makes provisions to maintain the compatibility and enhance usability with these resource-constrained server devices. As such, when the encryption/authentication fails, the BLE data and control messages can still be transmitted in plaintext as described in the specification (Figure 4). We believe that the contradiction originating from the details in the specification shown in Figures 3 and 4 may have confused the BLE stack developers. As such, they made the logical mistake of not aborting the connection in the event of failure of the encryption process which is mandatory for accessing

Table 1: List of BLE server devices examined to evaluate the impact of the design weakness-1.

Device Name	Support for link-layer authentication
Nest Protect Smoke Detector	×
Nest Cam Indoor Camera	×
SensorPush Temperature Sensor	×
Tahmo Tempi Temperature Sensor	×
August Smart Lock	×
Eve Door&Window Sensor	×
Eve Button Remote Control	×
Eve Energy Socket	×
Ilumi Smart Light Bulb	×
Polar H7 Heart Rate Sensor	×
Fitbit Versa Smartwatch	√
Oura Smart Ring	√

the security-sensitive attributes on a previously-paired server.

Figure 2b presents the details of how the BLESA succeeds against the client which employs the proactive authentication procedure by exploiting the design weaknesses along with the aforementioned implementation vulnerability. Specifically, after the connection to the adversary, if the client attempts to enable the encryption, the adversary sends an error message to the client specifying the non-availability of the long-term key so that the encryption fails. Nevertheless, the client does not follow the specification which recommends aborting the connection, but *continues the connection* with the adversary. Here, due to the first design weakness described in Section 4.2.1, the client then proceeds to send the read request for the target attribute in *plaintext*. Unlike the server, the adversary readily "grants" the access to the attribute, making it available at the lowest security level (i.e., security level 1, no security), and feeds spoofed attribute data to the client. As such, the client accepts the spoofed data from the adversary. This makes BLESA successful against the clients (e.g., Android and iOS devices) which employ the proactive authentication procedure during reconnection.

6 Implementation and Impact

In this section, we first examine the impact of the two identified design weaknesses in the real-world BLE stack implementations (Section 6.1 and Section 6.2). We then elaborate on the impact of BLESA on the Linux platforms (Section 6.3) and the Android/iOS mobile platforms (Section 6.4).

6.1 Examination of Design Weakness-1

To find out if there are indeed certain BLE devices that do not employ the link-layer authentication, we follow two approaches: (1) We inspect the behavior of a set of BLE devices' companion applications through static analysis. (2) We sample a set of real-world BLE devices and inspect their communication packets with runtime analysis.

Static Analysis of Mobile Applications. A typical BLE device (e.g., Fitbit fitness tracker) relies on its companion mobile application (e.g., the Fitbit application) which enables the end-user to access and manage the recorded attribute data (e.g., the number of footsteps). Since the pairing procedure (to establish a secret key) is one of the prerequisites for enabling the link-layer authentication, we can readily confirm that the BLE device does not support the link-layer authentication if there is no pairing API invoked in its companion application. To this end, we utilize the static analysis framework, FlowDroid [5], to inspect the Android BLE applications. More specifically, for each companion application, we use the Class Hierarchical Analysis (CHA) option in FlowDroid to build the call graph and find whether the application indeed invokes the pairing API from any of its entry points (e.g., an Activity). Since CHA is a relatively conservative way for call-graph construction, it may miss part of the method invocations. However, the analysis result is much more precise, with minimal false positives compared to other options (e.g., SPARK) [22]. Hence, it provides us a lower-bound of BLE apps using pairing.

Our static analysis starts with 33,785 popular applications, which are automatically crawled from AndroZoo website [1] in January and February 2020. Based on the constructed call-graph and corresponding APIs of these applications, we find that only 127 applications contain the BLE data read/write operations. We then check if the pairing API (`createBond()`) is invoked in these applications. Unfortunately, we find that only 41 (32.3%) out of the 127 inspected applications contain the pairing procedure, implying that a majority of the investigated BLE companion applications (67.7%) do not implement the link-layer authentication. Note that our analysis results already exclude those beacon applications which only scan for the BLE advertising messages, as such applications do not use the corresponding APIs to read/write the data stored on the BLE device.

Runtime Analysis of Transmitted Packets. We examine 12 BLE devices (as listed in Table 1) which are selected to represent a variety of application from mainstream BLE device manufacturers. We connect each of these server devices with a Google Pixel XL phone and read their attributes. During these experiments, we intercept the runtime communicated packets using an Ubertooth One radio [32]. By analyzing the intercepted packets, we find that 10 out of the 12 inspected BLE devices do not support any link-layer encryption/authentication. To this end, we conclude that most real-world BLE devices do not employ link-layer authentication.

Table 2: List of BLE client platforms examined to evaluate the impact of the design weakness-2 and utilized as the victims to evaluate BLESA.

Platform	OS	BLE Stack
Google Pixel XL	Android 8.1, 9, 10	Fluoride [16]
iPhone 6	iOS 12.1, 12.4	iOS BLE stack
Linux Laptop	Ubuntu 18.04	BlueZ 5.48 [13]
Thinkpad X1 Yoga	Windows 10 V. 1809	Windows Stack

6.2 Examination of Design Weakness-2

For the examination of the second weakness, we test four different client devices that cover all major platforms with different BLE stack implementations. We present detailed information about these devices in Table 2. We run experiments on each of these platforms to explore the answers for the following two questions: (1) Which of the two authentication procedures (i.e., reactive or proactive, as discussed in Section 4.2.2) does the client device utilize when it reconnects with a server device? (2) If the client device follows the proactive authentication procedure, does its BLE stack implementation have any logic flaw making it vulnerable against a spoofing attack? To answer these two questions, we first pair each tested client with a server (which is emulated using a Linux laptop, as shown in Table 3) and then disconnect them. Thereafter, we reconnect the tested client with the same server, prompt the client to read one of the server's attributes while capturing all BLE traffics via Wireshark [17].

By analyzing the traffic data corresponding to the Linux laptop used as the client, we find that the Linux BLE stack (i.e., BlueZ) implements the reactive authentication procedure. As a result, the Linux BLE stack suffers from the weakness as described in Section 4.2.2.

Meanwhile, we find that the Android, iOS, and Windows BLE stacks implement the proactive authentication procedure. Among them, the Windows BLE stack strictly follows the BLE specification. However, we discover that both Android and iOS devices continue the reconnection even when the encryption/authentication fails (as elaborated in Section 5.2).

Responsible Disclosure. We have reported the vulnerability to Apple and Google. Apple has acknowledged our findings, assigned the CVE-2020-9770 to the vulnerability, and fixed it. Although Google has also confirmed the vulnerability, but we have been told that our vulnerability report is similar to another report submitted three days earlier than us. Unfortunately, we confirm that the current up-to-date Android BLE implementation in our tested device (i.e., Google Pixel XL with Android 10) is still vulnerable.

6.3 BLESA against Linux Clients

For attacking the Linux client (the third row in Table 2), we utilize another Linux laptop (the first row in Table 3) as the server device. The emulated server runs a python script to provide a service corresponding to a sensitive attribute that can be read at security level 3 (i.e., with an encrypted and authenticated connection). To emulate the adversary, we use a Linux desktop with a CSR 4.0 [14] Bluetooth dongle (shown in Table 4). The adversary also runs a python script that handles the messages received from the client and launches BLESA against it. Besides, we use the `gatttool` [25] on the client device to send the attribute read requests and receive the responses.

Table 3: BLE server devices utilized to evaluate BLESA.

Platform	OS	BLE Stack
Linux Laptop	Ubuntu 18.04	BlueZ 5.48
Oura Ring [24]	Vendor OS	Vendor Firmware

Table 4: Adversary's platform for launching BLESA.

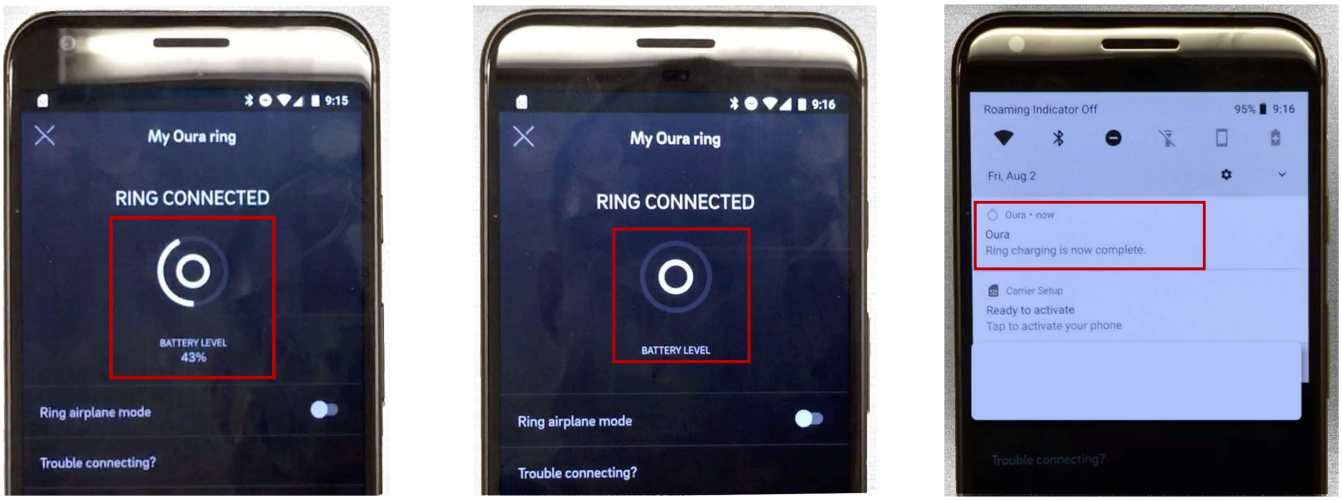
Platform	OS	BLE Stack
Linux PC & BT Dongle	Ubuntu 18.04	BlueZ 5.48

To launch BLESA, the adversary carries out the following steps which are akin to the procedure described in Section 5.1: ❶ scan (`bluetoothctl`) for the advertising packets transmitted by the server to record its MAC address; ❷ change the adversary's Bluetooth MAC address (`bdaddr` tool in BlueZ) to the server's MAC address so that the client can reconnect with the adversary; ❸ broadcast the same (impersonated) advertising packets as the server by issuing Host Controller Interface (HCI) commands, `HCI_LE_Set_Advertising_Parameters`, `HCI_LE_Set_Advertising_Data`, and `HCI_LE_Set_Advertising_Enable`, to the dongle; and ❹ inject the spoofed data via `ATT_READ_RSP` message after receiving `ATT_READ_REQ` message from the client. With these steps, the adversary successfully bypasses the reactive authentication procedure of the Linux client and tricks the client into accepting the spoofed data.

6.4 BLESA against Android/iOS Clients

Since BLESA bypasses the link-layer authentication by downgrading the connection to plaintext, all Android and iOS-based client devices communicating with the BLE server devices that do not employ any application-layer security mechanism (e.g., encryption or authentication) are vulnerable to BLESA. We note that according to the prior research [31], 46% of the Android applications (2,379 million cumulative installations) do not utilize application-layer security while reading data from BLE server devices. This implies that at least 46% of the Andriod applications are vulnerable against BLESA. The Apple app store is likely to have a similar proportion of vulnerable applications.

Here, we present how an adversary (shown in Table 4)



(a) Benign battery level (43%). (b) Spoofed battery level (0%). (c) Spoofed notification for charging.

Figure 5: Demonstration of the impact of BLESA on the data displayed by the Oura Ring application on the Google Pixel phone.

launches BLESA against a Google Pixel phone (the client device, shown in Table 2) by impersonating an Oura Ring (the server device, shown in Table 3). The adversary carries out the first three steps similar to those utilized for attacking Linux clients as discussed in Section 6.3, i.e., ❶ scan for the advertising packets of the ring; ❷ clone the advertising packets and the MAC address of the ring; ❸ advertise as the ring. Thereafter, the adversary performs the following: ❹ send an HCI command `HCI_LE_Long_Term_Key_Request_Negative_Reply` indicating the unavailability of the key to bypasse the encryption and authentication when the HCI event `HCI_LE_Long_Term_Key_Request` is received; ❺ inject the spoofed data via `ATT_READ_RSP` messages after receiving `ATT_READ_REQ` message from the phone. We note that the adversary can follow the same steps to launch BLESA against an iOS client device (the second row in Table 2).

With these steps, the adversary successfully injects spoofed data to the phone, and the companion application of the ring running on the phone accepts and displays the data in the application. Figure 5a shows that the correct/benign battery level of the ring is 43%. Through BLESA, we successfully inject the spoofed battery level (0%) to the application as shown in Figure 5b. At the same time, we also inject another spoofed message which triggers a notification in the application mentioning that the charging is complete as shown in Figure 5c. We note that although the first spoofed message tricks the application into believing that the battery level is at 0%, the application interestingly accepts another spoofed message (corresponding to charging completion) and displays the false notification to the user. A demo of this attack can be accessed at `https://paper-sub.github.io/`.

7 Mitigation of BLESA

To prevent BLESA, we need to secure the reconnection procedure between clients and their previously-paired server devices. We can achieve this by improving the BLE stack implementations and/or updating the BLE specification.

Updating Implementation. As far as the Linux BLE stack (BlueZ) is concerned, the client devices can be updated to only employ the proactive authentication. Also, the implementation vulnerability of the proactive authentication (discovered in this work) in Android and iOS clients can be mitigated by correctly following the BLE specification. The updated implementation must ensure that when the authentication with a previously-paired server fails, the client aborts the connection and restarts the pairing process. Now the more fundamental problem is that even for the proactive authentication, BLESA can bypass the link-layer authentication in the presence of other potential bugs in the implementation. This is a typical scenario why authentication at multiple layers are suggested by the security researchers [20]. While the vulnerability at the link-layer is being addressed, the server devices and their companion applications for the clients can employ application-layer authentication. Unfortunately, such an improvement might not be widely deployed because a large proportion of resource-constrained devices cannot be remotely updated.

Updating Specification. We argue that while it is important to fix the implementation bugs to promptly make the BLE devices secure against BLESA, it is equally critical to chart out a roadmap to prevent more advanced spoofing attacks. To this end, we should prevent the client from sending the attribute access request first and adjust the security level of the connection later based on the error message returned by the server. In other words, the client should first obtain the

authentic information about the access requirements of the attributes stored at the server, and then adjust the security level of the connection to meet those requirements before sending the access request. However, this approach requires the client to record the security requirements for each attribute on the server during the pairing process, We could also update the pairing process so that the server can send the security requirements of each of its attributes to the client.

Challenges and Future Directions. Patching different vulnerabilities at different layers in a timely fashion is a naive and straight forward approach to prevent spoofing attacks in BLE. However, this approach is unlikely to work well in practice because such ad-hoc, case-by-case fixes might not be feasibly adopted by the extremely fragmented BLE ecosystem at a large-scale. BLE device vendors tend to sacrifice security for utility, especially for low-end BLE devices. It is especially challenging to secure those low-end BLE devices that do not support any firmware update. Also, it is difficult to modify the pairing process while maintaining the backward compatibility with the legacy BLE devices. In addition, fundamentally addressing the BLE protocol vulnerabilities will require updating the BLE specification, which seems to be a non-trivial process. Therefore, such challenges bring the need for a device-agnostic, legacy-friendly, and comprehensive solution to mitigate the threat of spoofing attacks, as partially explored by a recent study [33].

8 Related Work

Formal analysis of the Bluetooth Protocol. Existing literature focuses on the formal analysis of Bluetooth pairing protocols. Chang et al. [12] have formally analyzed the authentication process in the Bluetooth legacy pairing and the secure simple pairing, and revealed their weaknesses. Phan et al. [29] have also formally verified and found vulnerabilities in the secure simple pairing. Arai et al. [4] have specifically analyzed the numeric comparison protocol used in the secure simple pairing and identified potential attacks. Nguyen et al. [28] have discovered vulnerabilities in the protocols that use out-of-band channels to establish shared keys. In contrast, in this paper, we focus on the formal modeling and analysis of the reconnection procedure in the BLE protocol, which leads to two design weaknesses in the authentication process.

Attacks against Bluetooth. A majority of the recent studies focus on the design weaknesses and implementation flaws in the Bluetooth stacks. Naveed et al. [27] have discussed the mis-binding between applications and external Bluetooth Classic devices. Sivakumaran et al. [31] have revealed similar mis-binding issues for BLE devices and analyzed the application-layer security issues in BLE applications on Android. Antonioli et al. [2] have discovered a logic flaw in the key negotiation protocol during pairing and exploited it to decrypt the encrypted traffic. BadBluetooth [34] has revealed how a malicious Bluetooth device can launch severe attacks against mobile platforms. Biham et al. [7] have focused on the Elliptic Curve Diffie-Hellman protocol and found weaknesses in the Bluetooth pairing procedure against invalid curve attacks. The study closest to our work is presented by Antonioli et al. [3] where they have presented impersonation attacks against Bluetooth Classic by exploiting the existing support for legacy pairing methods and role-switching feature. In this paper, we focus on the design weaknesses and implementation vulnerabilities in the BLE stack and how they can be exploited to launch the spoofing attack, BLESA. We highlight that since BLESA can be launched by an adversary without any additional assistance from a malicious application preinstalled on the client device, it is more powerful and practical than the attacks in the prior art [2, 3, 27, 31, 34].

9 Conclusion

In this paper, we carried out a formal verification of the reconnection procedure defined in the BLE specification, and we discovered two design weaknesses in the BLE link-layer authentication mechanism. By exploiting these design weaknesses, we proposed BLESA, a novel BLE spoofing attack through which an attacker can impersonate a BLE server device and provide spoofed data to a previously-paired BLE client device. BLESA can easily be launched against a BLE device running Linux.

In addition, further examination of the identified weaknesses in real-world BLE implementations revealed a related implementation vulnerability in the Android and iOS BLE stacks. Due to this vulnerability, these two stacks are vulnerable against BLESA. To showcase BLESA, we provided an in detail description of how to use this attack to spoof data coming from a fitness tracker to an Android smartphone. Moreover, we estimated the number of existing Android apps potentially affected by this attack. Finally, we discussed possible improvements in the reconnection procedure, to fundamentally mitigate the threat of spoofing attacks like BLESA.

Acknowledgments

We thank the anonymous reviewers for their valuable comments and suggestions. This work was supported in part by ONR under Grant N00014-18-1-2674. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of the ONR.

References

[1] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining*

Software Repositories, MSR '16, pages 468–471, New York, NY, USA, 2016. ACM.

[2] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. The KNOB is Broken: Exploiting Low Entropy in the Encryption Key Negotiation Of Bluetooth BR/EDR. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, pages 1047–1061, August 2019.

[3] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. BIAS: Bluetooth Impersonation AttackS. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2020.

[4] Kenichi Arai and Toshinobu Kaneko. Formal Verification of Improved Numeric Comparison Protocol for Secure Simple Paring in Bluetooth Using ProVerif. In *Proceedings of the International Conference on Security and Management (SAM)*, page 1. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2014.

[5] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 259–269, New York, NY, USA, 2014. Association for Computing Machinery.

[6] Vaibhav Bedi. Exploiting BLE smart bulb security using BtleJuice: A step-by-step guide. https://blog.attify.com/btlejuice-mitm-attack-smart-bulb/, 2018. Accessed: August 1, 2019.

[7] Eli Biham and Lior Neumann. Breaking the Bluetooth Pairing–The Fixed Coordinate Invalid Curve Attack. In *International Conference on Selected Areas in Cryptography*, pages 250–273. Springer, 2019.

[8] Bruno Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.

[9] Bluetooth Special Interest Group. 2019 Bluetooth Market Update. https://www.bluetooth.com/bluetooth-resources/2019-bluetooth-market-update/, 2019. Accessed: August 1, 2019.

[10] Bluetooth Special Interest Group. Core specifications 5.2. https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=478726, 2019. Accessed: April 1, 2020.

[11] Victor Casares. Mimo baby hack. https://medium.com/@victor_14768/mimo-baby-hack-ac7fa0ae3bfb, 2018. Accessed: August 1, 2019.

[12] Richard Chang and Vitaly Shmatikov. Formal analysis of authentication in bluetooth device pairing. *FCS-ARSPA07*, page 45, 2007.

[13] BlueZ contributers. Bluez. http://www.bluez.org/, 2019. Accessed: August 1, 2019.

[14] CSR. CSR 4.0 Bluetooth USB adapter. https://www.amazon.com/Bluetooth-Adapter-Songway-Computer-Keyboard/dp/B07KWVXBKZ/ref=sr_1_46?keywords=bluetooth+adapter+car+4.0&qid=1563227361&s=electronics&sr=1-46. Accessed: August 1, 2019.

[15] Aveek K. Das, Parth H. Pathak, Chen-Nee Chuah, and Prasant 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)*, pages 99–104, 2016.

[16] Android Developers. Fluoride Bluetooth stack. https://android.googlesource.com/platform/system/bt/+/181144a50114c824cfe3cdfd695c11a074673a5e/README.md, 2019. Accessed: August 1, 2019.

[17] Wireshark developers. Wireshark Go Deep. https://www.wireshark.org. Accessed: May 15, 2020.

[18] Danny Dolev and Andrew Yao. On the security of public key protocols. *IEEE Transactions on information theory*, 29(2):198–208, 1983.

[19] Fitbit. Fitbit versa 2. https://www.fitbit.com/us/products/smartwatches/versa, 2019. Accessed: Aug 1, 2019.

[20] Mario Frustaci, Pasquale Pace, Gianluca Aloi, and Giancarlo Fortino. Evaluating critical security issues of the IoT world: Present and future challenges. *IEEE Internet of things journal*, 5(4):2483–2495, 2017.

[21] Constantinos Kolias, Lucas Copi, Fengwei Zhang, and Angelos Stavrou. Breaking BLE beacons for fun but mostly profit. In *Proceedings of the 10th European Workshop on Systems Security (EuroSec)*, pages 4:1–4:6, 2017.

[22] Patrick Lam. Soot phase options (call graph construction). https://www.sable.mcgill.ca/soot/tutorial/phase/phase.html, 2019. Accessed: Aug 1, 2019.

[23] Angela Lonzetta, Peter Cope, Joseph Campbell, Bassam Mohd, and Thaier Hayajneh. Security vulnerabilities in Bluetooth technology as used in IoT. *Journal of Sensor and Actuator Networks*, 7(3):28, 2018.

[24] Oura Health Ltd. OURA SMART RING. https://ouraring.com/products/, 2019. Accessed: August 1, 2019.

[25] Ubuntu Manpage. gatttool - tool for Bluetooth Low Energy device. http://manpages.ubuntu.com/manpages/cosmic/man1/gatttool.1.html. Accessed: May 15, 2020.

[26] Tal Melamed. An active man-in-the-middle attack on Bluetooth smart devices. *International Journal of Safety and Security Engineering*, 8(2):200–211, 2018.

[27] Muhammad Naveed, Xiao-yong Zhou, Soteris Demetriou, XiaoFeng Wang, and Carl A Gunter. Inside job: Understanding and mitigating the threat of external device mis-binding on android. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, pages 1–14, 2014.

[28] Trung Nguyen and Jean Leneutre. Formal analysis of secure device pairing protocols. In *2014 IEEE 13th International Symposium on Network Computing and Applications*, pages 291–295. IEEE, 2014.

[29] Raphael C-W Phan and Patrick Mingard. Analyzing the secure simple pairing in Bluetooth v4.0. *Wireless Personal Communications*, 64(4):719–737, 2012.

[30] Polar. Polar h7 heartrate sensor. https://support.polar.com/us-en/support/H7_heart_rate_sensor, 2019. Accessed: Aug 1, 2019.

[31] Pallavi Sivakumaran and Jorge 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 USENIX Security Symposium (USENIX Security)*, pages 1–18, Santa Clara, CA, August 2019. USENIX Association.

[32] Ubertooth Developers. Ubertooth One. https://github.com/greatscottgadgets/ubertooth/wiki, 2019. Accessed: August 1, 2019.

[33] Jianliang Wu, Yuhong Nan, Vireshwar Kumar, Mathias Payer, and Dongyan Xu. BlueShield: Detecting Spoofing Attacks in Bluetooth Low Energy (BLE) Networks. In *23nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, 2020.

[34] Fenghao Xu, Wenrui Diao, Zhou Li, Jiongyi Chen, and Kehuan Zhang. BadBluetooth: Breaking Android Security Mechanisms via Malicious Bluetooth Peripherals. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS)*, pages 1–15, 2019.