BadBluetooth: Breaking Android Security Mechanisms via Malicious Bluetooth Peripherals

Fenghao Xu*, Wenrui Diao†‡, Zhou Li§, Jiongyi Chen*, Kehuan Zhang*

*The Chinese University of Hong Kong

Email: {xf016, cj015, khzhang}@ie.cuhk.edu.hk

†Shandong University, Email: diaowenrui@link.cuhk.edu.hk

‡Jinan University

§University of California, Irvine, Email: zhou.li@uci.edu

***Abstract*—Bluetooth is a widely used communication technology, especially under the scenarios of mobile computing and Internet of Things. Once paired with a host device, a Bluetooth device then can exchange commands and data, such as voice, keyboard/mouse inputs, network, blood pressure data, and so on, with the host. Due to the sensitivity of such data and commands, some security measures have already been built into the Bluetooth protocol, like authentication, encryption, authorization, etc.**

However, according to our studies on the Bluetooth protocol as well as its implementation on Android system, we find that there are still some design flaws which could lead to serious security consequences. For example, it is found that the authentication process on Bluetooth profiles is quite inconsistent and coarse-grained: if a paired device changes its profile, it automatically gets trust and users would not be notified. Also, there is no strict verification on the information provided by the Bluetooth device itself, so that a malicious device can deceive a user by changing its name, profile information, and icon to be displayed on the screen.

To better understand the problem, we performed a systematic study over the Bluetooth profiles and presented three attacks to demonstrate the feasibility and potential damages of such Bluetooth design flaws. The attacks were implemented on a Raspberry Pi 2 device and evaluated with different Android OS versions ranging from 5.1 to the latest 8.1. The results showed adversaries could bypass existing protections of Android (e.g., permissions, isolations, etc.), launch Man-in-the-Middle attack, control the victim apps and system, steal sensitive information, etc. To mitigate such threats, a new Bluetooth validation mechanism was proposed. We implemented the prototype system based on the AOSP project and deployed it on a Google Pixel 2 phone for evaluation. The experiment showed our solution could effectively prevent the attacks.

I. Introduction

As a wireless communication technology, Bluetooth has been adopted by a variety of electronic products including personal computers, smartphones and IoT devices, because of its technical advantages in short-range data exchange. Especially, given the context of IoT, smart devices could be connected with each other and controlled by a phone through Bluetooth.

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23482
www.ndss-symposium.org

Bluetooth has also become a lucrative target for adversaries due to its features like data sensitivity, transmission in the open air, and data handling in the kernel space. Recent years have seen lots of Bluetooth-related CVEs [11] resulting in system crashes, information leakage or privilege escalation on the target device. Besides the typical threats like data sniffing and weak pairing pass/PIN code, many vulnerabilities are caused by bugs in Bluetooth stacks, like kernel drivers, which could lead to code injections, arbitrary code execution, remote crash, etc [31], [38], [40]. There are also studies related to privacy issues. For example, Naveed et al. [34] discovered that an unauthorized app could steal sensitive data by connecting wrongfully to a third-party Bluetooth device.

To get a deeper understanding of Bluetooth security, we conducted a systematic study on Bluetooth at the logic level, including the underlying assumptions of the adversary model, device authentication, authorization, and security policies. Particularly, we focus on the Android platform due to its prevalence and its support of countless Bluetooth applications and services. In the end, we identified several new Bluetooth vulnerabilities even in the latest Android version. These vulnerabilities are mainly associated with *Bluetooth profile*, which is a standard interface about a particular Bluetooth functionality (e.g., audio transmission) but never thoroughly evaluated from the security perspective. For example, we found the current Android system assumes that a Bluetooth device only would support a fixed set of profiles, but this assumption is invalid because a malicious Bluetooth device actually can change its claimed profiles dynamically. As a result, several existing measures become insecure. For example, Android system will not check and notify users about the changes of device profiles, thus a device could first pair with the host using a benign function/profile and then switch to another profile and steal information without being identified. We also found that the Bluetooth device authentication is too coarse-grained and permissive, and most profiles, including the ones created dynamically, will be trusted by default once the user chooses to pair with that device. Even worse, the process of pairing with the device could be fully hidden to the user (see Section III for more details).

The newly discovered vulnerabilities can lead to severe attacks on user's privacy. To demonstrate the potential security implications of these vulnerabilities, we devise several concrete

attack examples under the name of *BadBluetooth*. In one attack, a malicious Bluetooth device could switch from a legitimate profile to the Human Interface Device (HID) profile stealthily. With such an HID profile, the malicious device could emulate the behavior of a Bluetooth keyboard and a Bluetooth mouse by injecting keystroke and mouse movements and click events. Consequently, it is able to change phone configurations, bypass security protections, and install malicious apps without being detected. In another attack, a malicious Bluetooth device could change its profile to Personal Area Networking (PAN) stealthily, then launch a Man-in-the-Middle attack to sniff the network traffic or inject spoofing packets (like DHCP/DNS replies pointing them to malicious servers).

Such vulnerabilities are not bugs caused by programming mistakes. Instead, they are rooted from the incorrect perception and assumptions on the Bluetooth communication. To mitigate the security threats, we design a new validation mechanism named *Profile Binding*. It enforces a fine-grained control for the Bluetooth profiles and prevents the unauthorized changes of profiles. We implemented and deployed our solution on Google Pixel 2. The evaluation result showed that it can prevent the *BadBluetooth* attack effectively with negligible overhead.

Contributions. We summarize the contributions of this paper as follows:

- *New vulnerabilities.* We investigated the design and implementation of Bluetooth on Android system and identified several vulnerabilities, such as the wrong assumptions on device profiles, coarse-grained device authentication and authorization mechanisms, as well as deceivable and vague user interface.
- *New Attacks.* To demonstrate the feasibility and security implications of our newly discovered vulnerabilities, we came up with several new attacks under realistic settings. These attacks can bypass existing data isolation mechanisms of Android, causing information leakage, changes of system security settings, etc. We implemented and evaluated them on different Android phones ranging from Android 5.1 to the latest Android 8.1.
- *Defense and Evaluations.* We proposed a fine-grained device profile management mechanism to mitigate the security threats. Also, we implemented it on Android 8.1 and demonstrated it could address the threats effectively.

Roadmap. The rest of this paper is organized as follows. Section II gives the necessary background about Bluetooth. Section III describes the Bluetooth design flaws found in our research. Section IV overviews the attacks against Android, and Section V describes these attacks in details. We evaluate our attacks under real-world settings in Section VI. Our defense solution is presented in Section VII. Section VIII discusses some advanced topics, and Section X concludes this paper.

II. Background

In this section, we introduce the relevant background about Bluetooth. We first overview Bluetooth stack and describe Bluetooth profile and connection mechanisms in details. Then, we describe how Bluetooth functionalities are supported by Android and how the risks are managed.

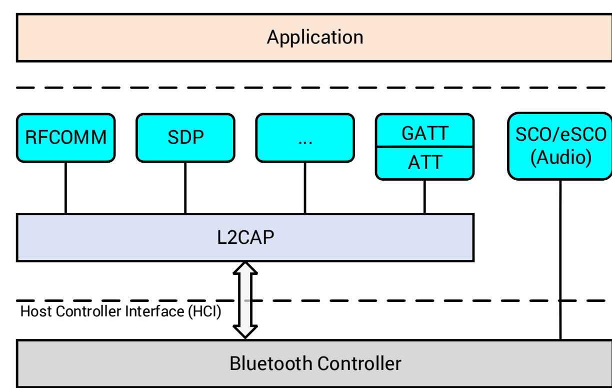


Fig. 1: Bluetooth Stack.

A. Bluetooth Overview

Bluetooth was proposed as a wireless technology standard to enable short-range data exchange, which was invented two decades ago. It has been gaining wide popularity among end-users: the forecast shows near 10 billion Bluetooth devices will be in use by 2018 [7], covering a variety of device types, including PC, mobile phone, smartwatch, car, medical appliances, etc. Comparing to another popular wireless standard, i.e., Wi-Fi, which is designed for wireless local area network (WLAN), Bluetooth is more user-centric, supporting wireless personal area network (WPAN) and requiring minimum configuration efforts. Currently, Bluetooth standard is managed by the Bluetooth Special Interests Group (SIG), and the latest specification is Bluetooth 5.0.

Bluetooth Stack. We illustrate the abstracted Bluetooth stack in Figure 1. In essence, Bluetooth stack is a multi-layer architecture including the lower physical and link layers, the middleware layer and the application layer [6]. The lower layers are implemented by Bluetooth chips, including radio controller, baseband controller, etc. They communicate with "host", i.e., the operating system running on the device, through Host Controller Interface (HCI). The protocols in the middleware layer are all implemented by the host. Different from other communication technology like Wi-Fi, Bluetooth protocols do not rely on the widely adopted TCP/IP stack. The base-level protocol for the middleware layer is *Logical Link Control Adaptation Protocol* (L2CAP), which can be treated as TCP for Bluetooth stack. It manages the connection between two Bluetooth devices, which implements features like QoS, flow-control, fragmentation and reassembly mechanisms. A suite of application-oriented protocols are devised on top of L2CAP. For example, *Radio Frequency Communications* (RFCOMM) is used to generate the serial data stream, which can replace the transmission of data over serial ports. *Service Discovery Protocol* (SDP) broadcasts the services (e.g., headset capability) supported by the host device and the associated parameters (e.g., device identifier) to other devices, in order to establish the connection. To enable more efficient data trans-

mission, audio transport can be supported using *Synchronous Connection-Oriented* (SCO) channel, without using L2CAP. The application layer defines the functionalities offered to users.

Starting from the Bluetooth 4.0, a technology named Bluetooth Low Energy (BLE) was incorporated, which aims to reduce the power consumption for new devices in healthcare and home entertainment. New protocols like *Generic Attribute Profile* (GATT) are included to facilitate BLE modes.

Bluetooth Profile. To regulate the communication between heterogeneous Bluetooth devices manufactured by different vendors, the concept of Bluetooth profile was proposed, which is characterized by a general functionality of a device. Each profile contains settings to bootstrap the communications, like the formats of user interface and dependencies of protocols. So far, there are more than 30 profiles standardized by Bluetooth SIG [10]. The most commonly used profile is Headset Profile (HSP), which specifies how a Bluetooth headset can be used with mobile phones. It relies on SCO channel to encode the audio and RFCOMM protocol to transfer AT commands [8] for control capabilities like answering a call. A device can claim a *subset* of profiles but the implementation must be compatible with the standard.

Bluetooth Connection. Before the connection is established between two Bluetooth devices, one device should be in the discoverable mode, which can choose to respond to an inquiry from the other nearby device with information like device name, device class, list of services (profiles) and technical information (e.g., manufacturer). Each device has a unique 48-bit MAC address but it is usually not used in the above process. Instead, a friendly name defined by the manufacturer or the user is displayed. However, if the inquiry initiator knows the address of another device, the inquiry has to be answered.

After the information is exchanged, a pairing procedure would be executed to authenticate the remote device and protect the communication against eavesdroppers. Pairing usually involves certain user interactions to confirm the identity of the remote device. Such a process could require a user to enter a PIN presented by the remote device or compare the numerical code on the displays of both devices, for example. If pairing is successful, a shared secret named *link key* is created to encrypt their communications, and both devices are said to be *bonded*. If both devices memorize the pairing information and the secret, they can connect to each other without going through pairing again in the future.

One thing to pay attention is that the communication for the two bonded devices is *profile-centric*: after retrieving necessary information from SDP, one device has to take additional step to connect to the profile of the other one before using its functionality (the first becomes initiator and the latter becomes acceptor). In addition, two devices can maintain multiple channels under different profiles. For example, a user's phone could connect to the headset profile and the keyboard profile of a single Bluetooth device at the same time.

B. Android Bluetooth

The early Android versions used Linux's BlueZ stack as its Bluetooth stack. Since Android 4.2, Google developed its own stack, named Bluedroid or Fluoride. For normal users, they could perform Bluetooth related operations through Android Settings (a system app). To interact with Bluetooth stack, both an Android third-party app and the Settings app could invoke `android.bluetooth` APIs to communicate with a system process, which is packaged as an app and located at `packages/apps/Bluetooth`. This system app implements various Bluetooth services and profiles. Receiving the request from the upper-level Android app, it further invoke into the native Bluetooth stack code, located at `system/bt`.

Bluetooth Permission. Since the Bluetooth communication may involve sensitive data, the access to the Bluetooth functionalities on Android is mediated by a set of permissions. A third-party app can initiate the discovery of nearby Bluetooth devices or change the Bluetooth settings if the `BLUETOOTH_ADMIN` permission is granted. Further, with `BLUETOOTH` permission, the app can perform Bluetooth communication with another device, such as requesting and accepting connections. The protection levels for both two permissions are `normal`, which means any third-party app claiming them will be auto-granted without reminding users. Since Bluetooth discovery may reveal the location of the user, from Android 6.0, if an app requests to scan nearby devices, it has to declare either the `dangerous`-level `ACCESS_COARSE_LOCATION` or `ACCESS_FINE_LOCATION` permission. By default, the pairing process needs the user's interaction. However, a system app can avoid this with a granted `signature`-level permission `BLUETOOTH_PRIVILEGED`.

Note that, the BadBluetooth attack described in this paper does not require any `dangerous`-level or `signature`-level permissions (see details in Section IV).

III. Design Weaknesses

The existing mechanisms around Bluetooth security focus on proving the identity of the remote device (through pairing), ensuring the confidentiality of the communication (through encryption), and restricting the capabilities of the untrusted apps on the host (through permission). These mechanisms work under the assumption that the remote device is trustworthy, say, its manufacturer or the owner certifies the device functionalities responsibly. However, such an assumption is not always true. More specifically, our study reveals that an adversary could manipulate the profiles on a remote device in an unexpected way and use it as a stepping-stone to attack the paired Android phone, casting severe threats to the phone owner.

This new problem rises mainly because the security model defined by the Bluetooth stack is coarse-grained, focusing on the device level. The problem is further complicated due to the issues underlying the design of the Bluetooth framework on the host, e.g., Android. Below we list five key issues and elaborate the potential security implications for each of them.

Weakness #1: Inconsistent Authentication Process on Profiles. Before two devices are bonded, a user could verify the identity of the remote device with an array of measures, like comparing the displayed PINs. However, the best practices regarding how profiles should be verified are not clear, since

they are never documented by Bluetooth core specifications. As such, the device and host vendors have to come up with ad-hoc ways for profile authentication, and mechanisms differ significantly among these vendors or even profiles. Taking Android as an example, the profiles are not listed during the pairing process and are only visible to the user and adjustable later (see Figure 2). If the device makes changes on the profiles, it still gets trusted since pairing has already done, and the user will not be immediately notified. Regarding how the profile channels are set up, some require user interactions (e.g., File Sharing) while some can be done silently through an app (e.g., Internet Access). As such, a device can contain adversarial functionalities without revealing them to the user in the beginning. For instance, a headset re-programmed by an adversary could enable Human Interface Device (HID) profile after being paired with a phone and send unauthorized keystrokes (see Section V-A).

Weakness #2: Overly Openness to Profile Connection. To better align with the Bluetooth specifications, a Bluetooth stack typically supports many profiles (e.g., 15 for Android 8.0 [2]). What's problematic here is that a pro-active approach is usually taken by the host, like Android: once the bond is created, the host will try its best to connect to all the profiles claimed by the remote device, without explaining the risk to the user or letting her vet the connections. Even though the user could disconnect certain profiles later in the device detail menu (see Figure 2), such a decision is not memorized by the host. The connections will be re-established when the devices are paired next time.

Weakness #3: Deceivable and Vague UI. When a user browses the list of paired Bluetooth devices, he could see the name and the icon of the device (example shown in Figure 2), which is given by the device during the pairing process. Though the information should be relevant to the core functionality of the device, there is no way to certify they are authentic. Previous research shows that a malicious device can choose the same name as another validated device's, to trick the user during pairing [34]. In this work, we found the icon can be manipulated as well for the same purpose. In fact, Bluetooth specification has defined a list of Class Device/Service (CoD) numbers [5] and each CoD number is associated with one icon reserved by Android. By changing the CoD number, the adversary can select the icon to be presented. Another issue with Android UI is the lack of Bluetooth-relevant information. For example, only two events relevant to Bluetooth are prompted in the notification bar: one showing that the Bluetooth of the host is turned on, and another showing that a remote device is connected. None of them reveals the status of profiles.

Weakness #4: Silent Pairing with Device. Pairing is supposed to have user to verify device identity, unless the bond has been successfully set up before. However, we found pairing can be completely hidden to the user even for the first-time setup. When pairing request is sent from the device side, Android system will pop up the pairing dialog for user confirmation. However, if the phone initiates this process, there might be no notification presented. Specifically, when the device has neither display ability nor input ability (e.g., headset), the pairing falls into "Just Works" mode [6], because both numerical

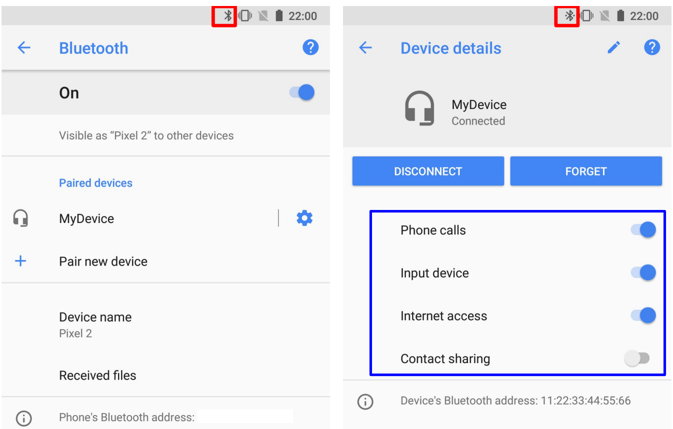


Fig. 2: Bluetooth Menu of Android (Google Pixel 2). The area in the red square shows the connection status - left one is disconnected, right one is connected. The area in the purple square lists the profiles currently in use, which can be adjusted by the user.

comparison or PIN input method become impossible. In this case, Android phone will not prompt to users. By manipulating the device configuration, this feature could be leveraged to pair with a Bluetooth device silently.

Weakness #5: No Permission Management for Profile. As described in Section II-B, Android restricts whether an app can access a Bluetooth device through a set of permissions. However, such a permission framework turns out to be too coarse-grained and mis-aligned with profiles. In particular, not all profiles are equally sensitive but which profile can be accessed is not regulated under the current permission framework. For instance, the profile regarding the Bluetooth keyboard (i.e., HID) should only be accessible to a system process. However, when a third-party app is granted `BLUETOOTH_ADMIN` permission, the keyboard becomes accessible automatically. Therefore, the app can further utilize the keyboard to inject inputs and take control of the phone and we demonstrate a working attack in Section V-A (the goal here is similar to the attack proposed by Fratantonio et al. [25] but our attack enables far more operations). So far, the protection on the critical profiles relies on removing the relevant code from the public APIs. However, we found a third-party app can still access those profiles through Java reflection.

Though the issue of coarse-grained Bluetooth permission has been mentioned by Naveed et al. [34], the focus is different. In particular, their work shows the permission does not prevent an unauthorized app (Bluetooth permission granted) to tamper the bond of an authorized device on the same phone. The issue we studied here regards profile.

IV. Attack Overview

The goal of our research is to explore and understand how the Bluetooth peripherals can gain high privileges and compromise user privacy in smartphones. Particularly, we focus on the Android platform due to its prevalence and its support of countless Bluetooth applications and services. In this section, we first introduce the adversary model in our

attacks. Then we describe the attack primitives and procedures. In Section V, we further explore how the attacks are achieved in real-world scenarios.

A. Adversary Model

In this study, we make two basic assumptions. We first assume a malicious app with Bluetooth permissions has been installed on the victim smartphone. Being granted with the Bluetooth permissions `BLUETOOTH` and `BLUETOOTH_ADMIN` which are the standard and common permissions for typical Bluetooth apps, the malicious app will be able to establish the bond and connecting the profiles with Bluetooth devices stealthily. Note that, since both `BLUETOOTH` and `BLUETOOTH_ADMIN` are just the `normal`-level permissions, the OS or Google Play will grant them to the malicious app without user confirmation. Therefore, this malicious app could be disguised as any type of apps, not just a Bluetooth app. As we will show later, such a malicious app can exploit the vulnerable designs in existing Android OS and Bluetooth devices and elevate its capabilities. For example, without requesting sensitive permissions or breaking the sandbox, it can capture the UI of other apps and steal sensitive information.

We also assume a Bluetooth device has been compromised and its firmware now contains malicious code. Adversaries can achieve this goal in several different ways. For example, they can first compromise the SDK of Bluetooth devices, which is similar to the attack of XcodeGhost [49]. Besides, Bluetooth devices may be hacked by previous owners, sellers or during the shipping process. What is more, the adversaries may be able to exploit the security weakness of Over-The-Air upgrading mechanism [22], especially with the help of the malicious app previously installed. We have studied the technical documents of popular Bluetooth chip-sets, including Nordic [37], Silicon Labs [32], TI [27], and found that their firmware verification is mainly to guarantee the transmission integrity (like CRC checksum, Hash values, etc), and there is no integrity check based on digital signatures. For CSR, another major Bluetooth chip-sets vendor, we do not have access their technical document [43], but according to messages from developers on GitHub [41], their OTA "*protocol seems to do challenge-response with a shared key rather than properly signing the firmware*", which might be insecure if adversaries could get the key via reverse engineering (and it would be left for future studies).

A large number of previous works on Bluetooth security focus on the vulnerabilities residing in the communication protocols and implementation of the software stack. Different to those works, we study the fundamental design flaws, which are much more difficult to fix.

To notice, previous works attacking the design flaw of Bluetooth stack or framework require the similar adversary capabilities [34]. In Section VIII, we will discuss more about the model and expansion of attacks.

B. Attack Procedure

Figure 3 illustrates the high-level attack procedures. We assume the malicious app is running at the background on user's smartphone. The attack could be launched when the screen is turned off, which indicates that the user is not around.

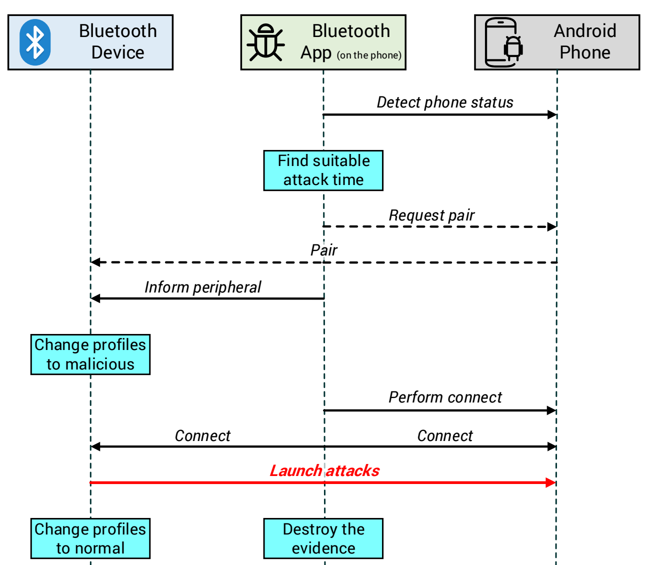


Fig. 3: Attack Flow. The existence of the dotted line depends. If the device is not paired yet, the app can request pair stealthily.

Then, the app creates the bond with the malicious device and set up the profile channel. After that, the app issues commands to the device to carry on the attack. We carefully design the attack flow to avoid any user interaction, making it hard to be observed.

Attack Primitives. We first describe the four attack primitives that enable our attack. The design weaknesses discussed in Section III are listed along with each primitive.

- **Changeable Profile (Weakness #1, #2, #3).** To hide certain profile from the user (e.g., HID profile), we instruct the device to add the profile after pairing, and remove it after the attack is completed. This could be achieved by programing the device to broadcast the service record related to the profile using SDP. Realizing the change of profiles is difficult from the user's perspective: the device detail menu (see Figure 2) would keep the same until the new profile is connected or the bond is reset. In addition, which profile is added is not shown in the notification bar.
- **Changeable Icon (Weakness #3).** Device icon is an important indicator to help user know the functionalities of the Bluetooth device. However, it can be easily changed when the device modifies the CoD number. Table I shows the icons we use to mislead user. The CoD number is composed of two fields. The first field describes the Major Device Class. And the second field describes the optional Minor Device Class or the Major Service Class, which can be set to all 0. If the CoD number is not recognized by Android, a general Bluetooth icon will be displayed.
- **Silent Pairing (Weakness #4).** Previous research [34] and app developers [14] also constructed the similar primitive. However, they rely on an

TABLE I: Bluetooth Device Icons.

Icon	CoD	Class Description
	0x100	Computer
	0x200	Phone
	0x404	Audio/Video-Wearable Headset
	0x418	Audio/Video-Headphones
	0x500	Peripheral
	0x540	Peripheral-Keyboard
	0x580	Peripheral-Pointing device
	0x600	Imaging
	0x000	General Bluetooth

API (`setPairingConfirmation()`), which is protected with the signatured permission `BLUETOOTH_PRIVILEGED` since Android 6.0. In contrast, our attack does not require such API use. As described before, we configure the remote device to work under the "Just Works" pairing mode. Then, the app can invoke just one API named `createBond()` using the identifier (MAC) of the device. Therefore, there is no manual confirmation involved.

- **Connecting Sensitive Profile (Weakness #5).** Our attack relies on exploiting sensitive profiles on the phone, which are supposed to be hidden from a third-party app. However, those profiles can be still accessed regardless of the protection. Normally, Android assumes the app creates a proxy class to operate on a profile, which encapsulates the IPC binder to the Bluetooth system process. The proxy classes of the sensitive profiles are not public, but when we program our app using *framework.jar* from a real phone instead of *android.jar* from Android SDK [1], we can directly use the non-public classes and methods, including the proxies of the hidden profiles. For example, as shown in the following code snippet, we invoke `getProfileProxy()` with a profile type `INPUT_DEVICE` as its parameter. If succeed, we can receive the proxy object whose class is `BluetoothInputDevice`. This class is non-public and has a method `connect()`. Our app could invoke this method to establish a channel to the input profile of the remote device.

```
1  final BluetoothDevice mDevice =
       mBtAdapter.getRemoteDevice("MAC");
2  private BluetoothInputDevice mProfile;
3  mBtAdapter.getProfileProxy(this,new
       BluetoothProfile.ServiceListenner()
4   { @override
5     public void onServiceConnected(int
          profile,BluetoothProfile proxy){
6      mProfile=(BluetoothInputDevice)
          proxy;
7      mProfile.connect(mDevice);
8   }
9  ...
10 },BluetoothProfile.INPUT_DEVICE);
```

Attack Phases. To deceive the user that the device is innocuous when pairing, the device can pretend to be a smart speaker or temperature sensor by using related icons and friendly names. The whole pairing process could also be completed stealthily. To communicate with the app, the device could use RFCOMM (regulated by Serial Port Profile), which is widely adopted by app-device communication and no profile will show up in the menu of device details. Below we elaborate our attack flow.

1) After the malicious app starts, it runs as an Android background service or a scheduled job. This service waits until the user is not around, by checking whether the phone screen is off through `PowerManager` or whether the time is at midnight.
2) If needed, the app will turn on Bluetooth through the API `BluetoothAdapter.enable()` and search for the other malicious device. After Android 6.0, such scanning process requires `ACCESS_COARSE_LOCATION` permission. However, it's not compulsory for our attack because the app could use the pre-stored device address directly and this will not result in Bluetooth scanning. As a result, the app could do silent pairing to stealthily create a bond.
3) Now the device is waiting for the commands from the app. Those commands are transmitted either through an in-band channel (e.g., Bluetooth RFCOMM channel) or an off-band channel (e.g., an Internet server connected by both).
4) Once receiving the commands, the device enables the attack-specific profile. Since the corresponding profile on the phone is sensitive, the app can use the profile connection primitive to establish the profile channel. In some scenarios, the device can take the first action to initiate the connection, and we discuss them in details in Section V. The malicious device and the app then leverage the capabilities of the enabled profile to attack the victim, like exfiltrating sensitive data.
5) Finally, the device could disable the used profile or terminate the connection. Moreover, the app could also unpair with the device using `removeBond()` API to avoid the victim's attention.

V. ATTACKS

In this section, we explore what kinds of attacks could be achieved under the general attack model described in the previous section. First we overview the supported Bluetooth profiles on Android and the ones we exploit. Then we demonstrate three types of concrete attacks through profile abuse.

Exploited Profiles. Table II lists the profiles supported by Android [2] with the corresponding usage scenarios. We do not list the transport-layer profiles like Bluetooth Network Encapsulation (BNEP), Object Exchange (OBEX) in this table since they are used to support other profiles and do not directly handle user information. In the list, Health Device Profile (HDP) and Serial Port Profile (SPP) are used to carry data for normal user apps, which can not be leveraged to attack the phone directly. Device ID Profile (DIP) helps SDP broadcast extra device information. The remaining profiles are evaluated for conducting our attack, and we identify three profiles which

TABLE II: Android Supported Profiles

Name	Description	Usage
HID	Human Interface Device	Keyboard
PAN	Personal Area Networking	Network Hotspot
HFP/HSP	Hands-Free/Headset	Wireless Headset
SAP	SIM Access	Car Kit
MAP	Message Access	Car Kit
PBAP	Phone Book Access	Car Kit
OPP	Object Push	File Transfer
A2DP	Advanced Audio Distribution	Wireless Speaker
AVRCP	Audio/Video Remote Control	Remote Media Controller
DIP	Device ID	Extra Device Information
HDP	Health Device	Blood Pressure Kit
SPP	Serial Port	App-specific

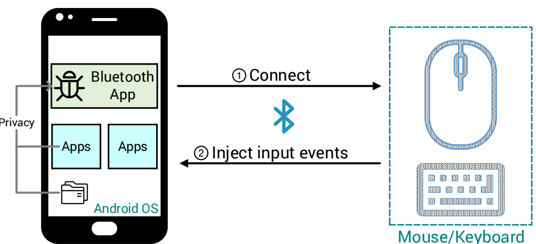


Fig. 4: HID Attack. The external device can inject input events. The malicious app could steal sensitive data with the help of the device.

could be leveraged – Human Interface Device (HID), Personal Area Networking (PAN), and Hands-Free/Headset (HFP/HSP). In the following, we introduce their capabilities and the attack scenarios enabled by abusing them.

Demo. The following attacks are demonstrated with demos posted at https://sites.google.com/view/bluetoothvul/. We used Google Pixel 2 equipped with Android 8.1 in the demo.

A. Human Interface Device

The Human Interface Device (HID) Profile enables the functionality of input devices like keyboard or mouse connecting to a phone. It is designed to facilitate the user to operate her phone with an external input device. For example, some people may project their phones to an external monitor and type text on it. With this capability equipped, a Bluetooth device is able to perform nearly any operations a real user can do on the phone. More specifically, Android provides the *fully functional keyboard and pointing device* (e.g., mouse) support through HID [3], and we can construct a input sequence equivalent to any user action (e.g., mouse click can be treated as user touch).

Figure 4 illustrates the flow of our attack. The Bluetooth device plays the acceptor role which is responsible for broadcasting SDP services. The installed malicious app initiates the connection process to connect the HID profile on the remote device. After the connection is established, this device gains full control over the input channel by sending HID reports. To notice, the input from the device is global to the Android phone, meaning that any running app and home screen can receive the input and handle it. As such, our attack breaks the app sandbox mechanism.

TABLE III: HID Report Format

Keyboard	Modifier Key	Regular Keys		
Mouse	Button Array	X Relative	Y Relative	Wheel

HID Report. When advertising the HID service, the SDP record contains a particular attribute - HID descriptor which tells the client (i.e., the phone) how to parse the payload packet. After the connection is established, the device could send a certain type HID report to generate a global input event on the phone.

In our attack, we leverage the HID descriptor to support standard mouse and keyboard functions on attack device. The corresponding HID report data format is shown in Table III. In a HID report, the header specifies the report type, and the remaining bytes are the payload. For keyboard data, the payload has several key bytes and one byte bit-array for modifier keys like “Right Control” key. For mouse data, the payload contains X-Y axis, wheel movement data and an extra button bit-array. Later on, we can specify these fields to perform our attacks.

Attack Strategy. Next, to construct a real attack, we still need to address some technical challenges. Below we describe the challenges and our strategies to tackle them:

- **Adaptive Attack.** To position the mouse precisely on the targeted item, a challenge here is to determine the position of mouse pointer. Since different phone brands and Android versions usually have different UI layout, in attack phase 3, the malicious app will also collect the UI information via `android.os.Build` and notify the device to activate the matched payload. On the other hand, the attack device itself can also retrieve phone related information (e.g., phone brand) through the SDP record of the phone. Moreover, due to the uncertainty of initial pointer position, we move the pointer to left-bottom as the origin point by sending enough mouse movement reports.
- **Input Capability.** By constructing the HID input report, we can freely move the mouse or inject a key event on the phone. What’s more, we found that Android defines various functional keys [4] like “Home”, “Back”, and “Volume Control” besides normal letters. So it is possible to utilize these keys to enhance our attacks. In detail, when the phone receive the HID report, it will first parse the report payload into a Linux input event based on the previous provided HID descriptor. Then, for keyboard, there exists an mapping relationship between Linux input key code and Android defined key event, which can be found in `/frameworks/base/data/keyboards/Generic.kl` of Android source code. We then adjust our HID descriptor to enable these special keys usage. We summarize some functional keys which can be applied in our attack in Table IV [4]. What is more,

TABLE IV: Android Functional Keys

Linux Key Code Name	Description (effect on Android)
KEY_ENTER	Enter Key (click)
KEY_TAB	Tab Key (select item)
KEY_SYSRQ	Screenshot
KEY_COMPOSE	Menu Key (open menu for current app)
KEY_POWER	Power Key (open/close screen)
KEY_WWW	Explorer (launch browser app)
KEY_PHONE	Call (launch phone app)
KEY_MAIL	Envelope (launch mail app)
KEY_ADDRESSBOOK	Contacts (launch phone book app)
KEY_HOMEPAGE	Home Key
KEY_BACK	Back Key

common shortcuts like copy (*KEY_CTRL+KEY_C*) and paste key (*KEY_CTRL+KEY_V*) combination are available as well. And we also found that even without the mouse capability, we can simulate the moving or clicking task by sending *KEY_TAB* to select a certain item on the screen and *KEY_ENTER* to perform the click operation. This approach could make our attack stealthier and quicker.

- **Output Capability.** Keyboard and mouse only provides input ability. However, if the attacker wants to do more advanced attacks, output ability is necessary. In other words, if we can obtain the view of phone's UI, we can simulate full interaction capability of a user. Indeed, we found that there is a key named *KEY_SYSRQ* which stands for screenshot in the standard key code scheme, which will truly capture the phone screen on Android. Thus, we can inject this key to Android phone resulting in generating a global screenshot. Besides, another way to capture output is to select the texts on some views and then send "copy" shortcut to copy the text. Next, the app can read the text from system clipboard by using `ClipboardManager`. The limitation of this method is that not all the texts are selectable and the information can be gathered is much less than a screenshot image.

As a result, with these abilities, the attack could introduce severe consequences to the victim. We summarize them with three high-level categories as follows.

Attack: Information Stealing. Since we can capture screenshot globally, which can cover any foreground application in the screen, we can steal very sensitive information from normal or system app like private emails and messages, phone books, etc, and send them out of the phone. For example, we can grant our app the `WRITE_EXTERNAL_STORAGE` permission using the input ability and fetch the screenshot then send them out via Internet (a normal permission). Or we can use input ability to transfer them through another app like Web Browser (open a malicious uploading website) or Email. Finally, the app can delete the screenshot to destroy the evidence.

Attack: App and System Controlling. Most security mechanisms on Android phone are enforced with user's involvement. For example, after Android 6.0, all `dangerous`-level permissions should be granted at runtime by user confirmation. And many security and privacy policies are controlled by the system settings. There is no way for a normal app to modify the critical settings or perform a cross-app operation. However, by equipping with an external HID device, we can arbitrarily control what we want just like normal user interaction.

For example, we can grant all the dangerous permissions to our app thus causing continuous damage when the device is disconnected. And we can invade other apps by force stopping, uninstalling, or injecting input events on them. Moreover, we are able to install another malicious app. Modifying the critical system setting preferences is easy as well. Before the attack, we could choose the proper payload which contains the UI layout and item position information based on the Android version and phone brand. However, the user may personalize their phone and legitimate apps may have various appearance. To handle this problem, we could use the previous attack to get the screenshot and perform the image analysis locally or remotely to get the precise layout in order to attack them accordingly.

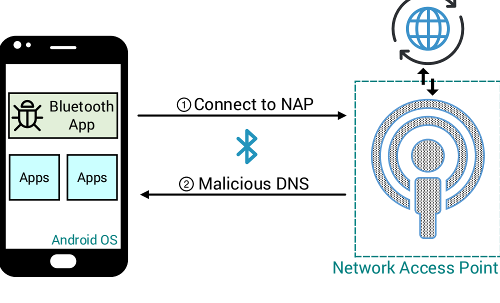
And through our experiment, we could even shutdown or reboot the phone by simulating the click of the power button. In detail, if we send *KEY_POWER* and wait a short period till sending the button release event, which simulates the long press of power button, the power manage menu will pop up. After that, we can select the shutdown or restart menu item.

Attack: Beyond the Phone. Besides being the interface to help the user process the daily tasks, the phone can also be used as data vault, keeping user's identity information or storing the token for many applications. Therefore, if the attacker takes control of the phone, he may steal stored token like a verification code in a text message or log himself into a website through remembered password. He may also abuse the victim identity like sending spam emails. He can even open the camera and capture the surrounding environment thus severely breach the victim's privacy.

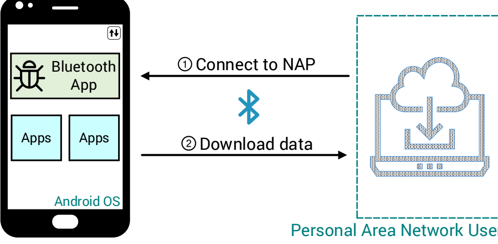
Limitations. Some attack operations, like capturing the screenshot of foreground apps, will fail when the phone is securely locked (for locking without PIN/Pattern, the attack still works through simulating swiping screen). Though our attack can inject keyboard and mouse input, unlocking the phone would be impossible if the user chooses PIN code that we do not know beforehand or enables other strong login mechanisms. A subset of operations are still effective under the locking scenario, like powering off the phone and turning on the camera. To notice, the following two attacks are still effective even when the phone is securely locked.

B. *Personal Area Networking*

Next, we investigate how the network communication can be tampered by exploiting the Personal Area Networking (PAN) profile, which manages the networking functionality through Bluetooth channels. This profile relies on BNEP protocol and defines 3 roles - Network Access Point (NAP), Group Ad-hoc Network (GN), and PAN User (PANU). A common use case is that one device who has an additional network resource like smartphone can act as a NAP to forward



(a) Device as NAP



(b) Device as PANU

Fig. 5: PAN Attacks. Figure (a) shows that the device can sniff and spoof traffic of the phone. Figure (b) shows the device can consume the host network bandwidth without permission.

Ethernet packets and provide DHCP service usually at the same time. The other device will be the PANU to share the network bandwidth of the NAP. Both roles are supported by Android but there is no protection mechanism in place to prevent a malicious app or device from abusing these roles. We construct two attacks as shown in Figure 5.

Attack: Network Sniffing and Spoofing. Since the phone could access Internet via the Bluetooth device, it is possible to provide the NAP service on the device side and do the network Man-in-the-Middle attack. In this attack, we enable the standard NAT service on the device and wait for the connection from the phone. Once the phone is connected, the Bluetooth device would receive all the Ethernet packets carried by BNEP from the phone and pass it to a pre-build virtual bridge. The bridge can then forward the traffic to a remote entity if the device has its own Internet access component. Then we can intercept all the traffic on that bridge. Note that accessing Internet can be achieved via a embedded sim card (cellular network), wired or wireless Internet connection of the device itself. Many IoT devices like smart speakers have built in such capability. And for the case that the device itself cannot access the Internet, it can still sniff a part of traffic like login request which contains sensitive information.

After establishing the Bluetooth network connection, the phone (PANU) will query for the networking settings from the NAP. The DHCP server on the virtual bridge can listen for this query and return a malicious DNS server address. This DNS server could be a public server owned by the attacker or just built upon the device.

A mechanism we want to mention here is the network resource priority on Android. As we know, the Android phone can use Wi-Fi and cellular network to access Internet beyond Bluetooth. So if multiple network sources appear, Android will automatically choose one through an internal ranking scheme. Through our investigation, we found Bluetooth network has the highest base score than other frequently used network types (Wi-Fi and cellular data). What is more, Android will conduct a connectivity testing (e.g., ping a google website) before the final decision and deduct points if the testing fails. So we can easily manage the network to select our Bluetooth NAP as long as the testing is passed, which naturally holds if the device owns Internet access ability.

The whole process can be done in the background even when the phone is securely locked. And we noticed that even when the phone is unlocked and used by the user, our attack only introduces an inconspicuous change in the notification bar, if a Wi-Fi connection has been established (a small mark on Wi-Fi icon). If the phone does not use Internet initially, we can enforce it as well. In summary, through this attack, we can force all the Internet traffic on the phone to go through our device. As a result, we can intercept sensitive information or do the spoofing attack.

Attack: Network Consumption. From another angle, the phone can also act as a NAT and share its network resource via Bluetooth. So in this attack, the device claims its identity as a PANU and try to connect and share the phone's network. Ideally, Android ought to forbid such connection by default and require user interaction. In reality, opening the Bluetooth tethering could be easily done by an app without any privileged permission granted. The API we used is `setBluetoothTethering()` of `BluetoothPan` class. To notice, this setting is global which is effective for all the external devices as well. Again, this implies the problematic implementation of Bluetooth management on Android.

As a result, once the app enables that setting, the device can try to connect to the phone NAT. With that, the device could send out collected information or receive data for firmware updating. Besides, the device can consume the network maliciously to cause extra data usage.

C. Hands-Free

Bluetooth supports audio transmission in two means. As shown in Section II, the first one is to transfer the audio signal natively by SCO channel. The latter one utilizes packets to distribute the audio data (Advanced Audio Distribution - A2DP). Headset Profile (HSP) and Hands-Free Profile (HFP) are two typical profiles relying on SCO channel, while we focus on HFP since it supports more features than HSP and has been widely adopted nowadays. A headset device implementing HFP allows user to perform operations (e.g., make phone calls) by issuing the commands without touching the phone. Also, the device could receive the telephony audio and answer phone calls using HFP. Therefore, when a malicious device implements HFP, it will be able to manipulate the audio input and receive the audio output of the phone. Figure 6 shows how an attacker can abuse these profiles to compromise user's privacy.

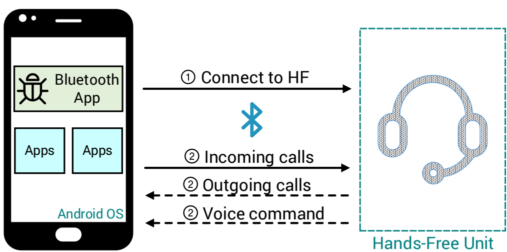


Fig. 6: HFP Attack. After connection, the Bluetooth device can control the incoming and outgoing calls. Also, it can inject voice command if Google Assistant is enabled.

Attack: Telephony Control. HFP defines two roles - Audio Gateway (AG) and Hands-Free Unit (HF). AG like a cellular phone can transfer the telephony status and open SCO connection for streaming the voice to HF (typically a headset). And the HF could issue several commands like accepting, rejecting an incoming call or terminating the current call, etc. In this attack, the device will claim the HF role, and wait for connection from the phone. Initially, AG and HF will establish a RFCOMM channel to exchange the handshake message and phone status using various AT Command. Then based on the telephony situation, the device may send command to answer, reject or terminate an incoming call. What is more, the device is able to dial arbitrary number resulting in a outgoing calls.

All the functions mentioned above could be done on a locked Android phone. Under our attack model, we can successfully force the phone to connect to the HFP-enabled device thus taking over the telephony function. For example, the device can record an incoming call and answer with prepared voice file.

Attack: Voice Command Injection. Besides the telephony function, we found the HFP can also trigger the Google Voice Assistant. And by default, this Google service will allow Bluetooth headset to send voice command even when the phone is securely locked. In the attack, we first trigger the assistant and open the audio connection. Then we can inject any voice command it supports. However, we found the voice feedback is carried by A2DP rather than HFP SCO channel. So the device could claim the A2DP profile at same time and once connected, the phone will send the voice feedback to the device. As a consequence, the attacker is able to inject commands and steal information through the voice channel stealthily.

D. Other Profiles

Besides the profiles we tested above, SIM Access (SAP), Message Access (MAP), Phone Book Access (PBAP) and Object Push Profile (OPP) are potential targets. However, those profiles require the Bluetooth device to be the initiator and the phone to be the acceptor, which is opposite to the attack flow described before (see Section IV-B). As a result, the user will be notified when the Bluetooth device requests to connect under those profiles and the request has to be approved manually, making the attack less stealthy.

Fig. 7: Experimental Devices.

VI. Implementations and Evaluations

In this section, we introduce the technical implementations of our attacks and discuss its scope.

Hardware Setup. We used a Raspberry Pi 2 (900MHz quad-core ARM CPU with 1GB memory) as the attack device, as shown in Figure 7. It runs Raspbian, a customized Linux OS. Also, a CSR8510 Bluetooth USB dongle is attached because no built-in Bluetooth chip is on Raspberry Pi 2. In practice, a bare-metal device equipped with low-cost Bluetooth chip [9] is sufficient to launch our attacks. The smartphone used as the host is Google Pixel 2 equipped with Android 8.1.

Implementations. We implemented a prototype of attack program (for Raspberry Pi 2) with around 1,100 Python lines of code. Our implementation was mainly based on the PyBluez [16] package which encapsulates the build-in BlueZ [12] (Linux Bluetooth protocol stack) of Raspbian and could manage the system Bluetooth resources. The main feature provided by PyBluez is to establish an L2CAP or RFCOMM connection. Also, some open-source softwares or libraries are integrated into our attack program for specific purposes. In details, the HID attack was implemented utilizing raw L2CAP channel directly. To the PAN attack, tcpdump [18] and dnsmasq [13] are used to sniff network traffic and set up DHCP/DNS servers. In the HFP attack, we used pulseaudio [15] to handle the audio processing and ofono [17] to verify the feasibility of this attack. In the real attack, we used raw RFCOMM to achieve it.

Besides, a malicious Android app is needed to assist launching the BadBluetooth attack. Its functionality is simple, mainly for connecting to the Bluetooth device. We invoked Android hidden APIs to implement such a requirement, as illustrated in Table V.

TABLE V: Attack Implementations on Android

Attack	Invoked APIs
HID	`BluetoothInputDevice.connect()`
PAN	`BluetoothPan.setBluetoothTethering()` `BluetoothPan.connect()`
HFP	`BluetoothHeadset.connect()`

TABLE VI: Attack Results

Phone Brand	OS	Vulnerable
Google Pixel 2	AOSP Android 8.1	Yes
Google Nexus 6	AOSP Android 7.1	Yes‡
Google Nexus 6	AOSP Android 6.0	Yes‡
Sony Xperia XZs	Sony Official Android 8.0	Yes‡
Samsung Galaxy S7	Samsung Official Android 7.0	Yes‡
Huawei P10	Huawei Official Android 8.0	Yes‡*
Meizu M3 Note	Meizu Official Android 5.1	Yes‡*

‡:Exclude Network Consumption Attack
*:Exclude Voice Command Injection Attack

Scope of Attacks. To evaluate the scope of our attacks, we selected the other 6 Android phones equipped with different Android OSes (from Android 5.1 to the latest Android 8.1) and tested the attacks against them. In our experiment, Google Voice Assistant is only available on the phones equipped with Google Service Framework (GSF). Therefore, the voice command injection attack was not tested on Huawei P10 and Meizu M3 Note. Besides, we found the `WRITE_SETTINGS` permission is needed to launch network consumption attack except Google Pixel 2 (Android 8.1). Except for the above two attacks, all the other attacks were successfully launched on all phones as listed in Table VI.

VII. PROFILE BINDING FOR ANDROID

The design flaws of the Bluetooth stack and the BadBluetooth attacks described in this paper should be fixed timely. In this section, we propose a lightweight solution named *Profile Binding* for Android, which provides a fine-grained control for the Bluetooth profiles and better visibility of profiles to user.

A. Overview

The high-level idea of our protection mechanism is to enhance the control of Bluetooth profiles and prevent the unapproved changes of profiles. In particular, we bind the device with a permitted profile list and prohibit other profile connections. In practice, when processing a pairing request, the system will prompt a selection list containing the advertised profiles of the external Bluetooth device for the user to approve. After that, the system will create a *binding policy* based on the user's selection, and further mediate every profile connection intent to enforce the policy checking.

As a result, this mechanism could let user vet the device profile and prevent unnoticed profile changing. Meanwhile, the silent pairing weakness is immediately addressed since the pairing process could not be hidden to the user anymore.

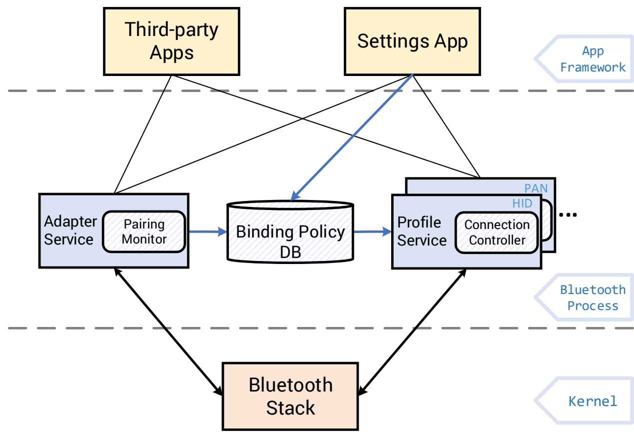


Fig. 8: Overview of the Profile Binding mechanism. The black lines show the original communication flow, while the white blocks and blue lines represent our defense framework.

Architecture. Figure 8 illustrates the updated architecture of the Android Bluetooth subsystem that deploys our defense framework. This framework contains three main modules: *Binding Policy DB*, *Pairing Monitor* and *Connection Controller*. In the original design of the Bluetooth subsystem, any upper-layer apps including Settings app (under user's control) could communicate with the Android Bluetooth process through IPC requests. For example, an app can initiate the pairing using `AdapterService.createBond()` or establish the profile connection through various `ProfileService`.

After deploying our defense, any unauthorized pairing and connection intent will be prohibited. The pairing monitor module integrated into `AdapterService` will create binding policies for Bluetooth devices. Then the connection controller module performs a policy validation in each `ProfileService`.

Note that, all three defense modules are integrated within the Android Bluetooth process, which ensures every pairing or connection intent from upper layer will be checked. All Bluetooth Java APIs regarding pairing and profile connecting will finally fall into this system process. Though there exist native Bluetooth functions like `createBondNative()` and `connectHidNative()`, it is still impossible to bypass our defense through native code. According to our investigation as well as mentioned in [34], only the Bluetooth process has the privileged permission to access the underlying Bluetooth stack directly, which is protected by the Linux user-based access control mechanism.

Workflow. The defense is implemented around the binding policy which is generated in the Bluetooth pairing phase. As described before, either the phone or the device could initiate the pairing. For the former case, both third-party apps and the user (through Settings app) will finally invoke the API `AdapterService.createBond()` with the target device's MAC address. In the latter case, when the Android Bluetooth process receives an external pairing request, the callback function `sspRequestCallback()` or

`pinRequestCallback()` of `AdapterService` will be called.

For both pairing cases, our defense framework will pop up a dialog showing the profiles declared by the Bluetooth device (extracted from its SDP records). After the user selects the permitted profiles manually, a policy record which binds the user's choice (a profile list) with the device (MAC address) will be inserted into the policy database. Therefore, our scheme supports the user to vet the device explicitly and prevents the silent pairing behavior. The policy associated with each device will be validated whenever `ProfileService` receives a connection intent. If the profile type indicated by the connection appears in the policy record of the target device, this connection request will be approved and sent to the Bluetooth stack. Otherwise, this connection request will be rejected. As a result, the BadBluetooth attack will be prevented because a malicious device could not hide or change its profile without user permission.

B. Implementation

Our proposed defense solution could prevent the BadBluetooth attacks and address the current Android Bluetooth weaknesses in the meantime. In the following parts, we describe the improvements to each weakness and the corresponding technical implementations of each module.

Pairing Monitor (Weakness #1, #2, #4). The pairing monitor module inspects both the incoming and outgoing pairing requests. Then it fetches the device SDP to generate the profile candidate list. After that, as shown in Figure 9, it presents a multi-choice dialog for user confirmation. We also remove the original system dialog (if it exists) and merge with ours to enhance the user experience. Finally, we save the permitted profiles as a bitmap associated with the device using the `Settings.Secure` storage, which cannot be modified by third-party apps. Through this approach, we prevent silent pairing and provide a fine-grained control method for users.

Connection Controller (Weakness #1, #2, #5). This module locates profile by `ProfileService` to enforce the policy validation. We adopt the whitelist approach to restrict the connection. Specifically, only if the device (MAC address) is registered in the policy database and the desired profile is set to be allowed, this connection could pass through. Otherwise, it gets denied immediately. Moreover, to unpair a device, the `Adapter.removeBond()` will be invoked. In this case, we will remove the device policy record accordingly.

Settings App (Weakness #1, #3). To enhance the usability for users, we also create the `updateProfile` method on the policy database and only expose it to the Settings app (protected by privileged permission). Therefore, the user could adjust the profile preference (binding policy) later. Moreover, to provide more meaningful information and reveal potential risk, we modify the Bluetooth icon mechanism. In our scheme, the device icon is chosen by its "behavior". Specifically, it is always the job of supporting profiles of a device to determine the icon. If a device claims more than one profile, the most "dangerous" one will be presented. We define the danger level as: HID > PAN > HFP > Others.

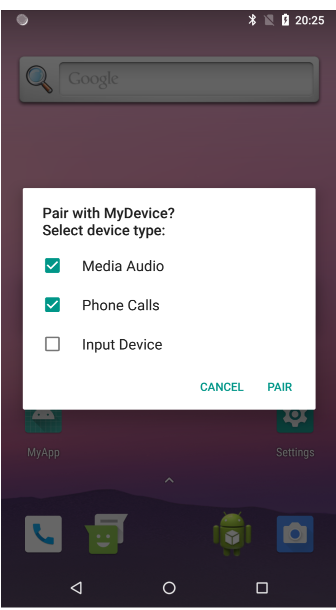


Fig. 9: Pairing dialog example of our defense. This dialog is shown when a pairing process happens.

C. Defense Coverage

As discussed in Section V-D, for some profiles like OPP, the device may initiate the connection without broadcasting in SDP. It is out of the scope of our defense, because, in the original mechanism of the Android Bluetooth stack, it will notify the user appropriately. Alternatively, our scheme can unify all profiles by showing them together at the pairing (no matter the device claims or not). However, such design is not user-friendly as a long list will be shown to the user every time. So, we did not follow this approach.

D. Evaluation

To evaluate the defense effect and corresponding overhead, we conduct several experiments on Google Pixel 2, which has a 2.35 GHz processor and 4GB memory with our modified AOSP Android 8.1.

Effectiveness. To examine the effectiveness of the profile binding mechanism, we launched all the attacks described in Section V on the phone. We found that all the pairing process is monitored and prompted to users, and only explicitly granted profiles can be connected. Therefore, the BadBluetooth attack is completely mitigated by our defense framework.

Performance. Pairing to an external device is adjusted to be noticed by the user, and our system should not cause prominent delay of UI-transition. In the meantime, the performance of normal operations should not be impacted. Given that the policy validation is supposed to be most time-consuming among all the introduced components, we instrumented the `connect()` methods and measured the execution time delay for certain profile connection (HID, PAN, and HFP). Our measurement process excludes the native function execution. For each profile connection, we conducted the test 10 times. The results are shown in Table VII. We can see the delays

TABLE VII: Profile connection evaluation. (mean/std)

ProfileService Class	Original (μs)	Defense (μs)	Delays (μs)	Total* (μs)
`HidService`	494.9/63.0	605.5/49.0	110.6	2546.0/589.4
`PanService`	235.8/45.8	460.4/43.1	224.6	1890.5/420.5
`HeadsetService`	473.5/62.4	522.2/66.5	48.7	2359.3/326.1

*:From upper-layer API call to connection completion (original Android OS).

are from 48.7μs to 224.6μs, which is hardly perceivable. Comparing with the total time cost (from upper layer API calls to connection completion), the delay is less than 12%.

VIII. Discussion

We first discuss the limitations of our attack in this section. Then we describe other adversary models to be considered to expand BadBluetooth attack. In addition, we believe the weaknesses we discovered are not just limited to a single device or a single OS. Therefore, in the long run, the protection should not only rely on the platform-specific implementations but also need to reconsider the design of Bluetooth stack.

Responsible Disclosure. Before the submission, we have reported our findings to the Android security team responsibly. They acknowledged the problems and are developing the corresponding solution. We will work further with them to better understand the issues underlying Bluetooth design and develop new defense mechanisms accordingly.

Limitations and Extensions of BadBluetooth. Our adversary model requires both a malicious device and a colluded app to successfully launch the BadBluetooth attack. Here we discuss other scenarios when some components are not controlled by attacker initially. If we assume only the malicious app is installed on the victim smartphone, then the app is able to discover and exploit nearby devices through Bluetooth channel. For example, a vulnerable Bluetooth device (e.g., has Bluetooth driver or application code bugs) may be compromised to install malicious profiles or remotely controlled by the app. In another example, the firmware updating process could be leveraged to compromise a Bluetooth device (as mentioned in Section IV). As a result, the attack is still feasible when there are vulnerable Bluetooth devices.

We implemented the attacks using Raspberry Pi 2, a dedicated device as the Bluetooth peripheral. However, we found the host like smartphone itself could also be programmed like an external device since its underlying Bluetooth controller has the same capabilities. The difference is that an app located in the user space can only access limited APIs provided by the OS. Through taking advantage of smartphone Bluetooth stack, attacks are still possible without a physical device, which leads to a more general threat model. For example, we note that future Android version plans to bring the HID device ability to normal app [26]. As a result, an app might be able to make a phone behave like a mouse or a keyboard. Therefore, it brings the risk that a malicious app controls its host phone to attack another connected phone through the Bluetooth channel.

Future Directions. Firstly, we plan to investigate the attack feasibility of above mentioned app-based BadBluetooth. Besides, Bluetooth firmware updating is also an appropriate entry point to study Bluetooth security. We will consider them as future directions.

In this work, we mainly focused on Android platform. However, the exposed weaknesses and problems may still exists on other OS platforms like iOS, Windows or Linux. For example, through a preliminary study on these platforms, we found none of them provides a good solution to the UI issue. Windows relies on the CoD number instead of the real profiles. iOS does not display icons when scanning nearby devices, and some Linux versions use a unified Bluetooth icon for all types of devices. Therefore, users may have to face difficulties in figuring out the real identity of a Bluetooth device. What is more, the profile authentication problems could still exist due to the vague Bluetooth specifications. Similar to the attacks on Android, a malicious insider could leverage the potential flaws to launch attacks. The complete attack implementations rely on the specific Bluetooth resource management mechanisms on different platforms. Therefore, it would also be a potential research direction to study in the future.

Bluetooth Design. Though the concrete attacks could be mitigated, the fundamental design weaknesses discovered in this paper cannot be addressed by Android itself. We believe these design weaknesses should be fixed on Bluetooth specifications in the long run. Looking through the whole Bluetooth specification, it puts many efforts to the functional diversity, transmission performance, and so forth. However, the security requirement is neglected to some extent and mainly relies on the implementation of device vendors.

We believe it is not a correct understanding that the Bluetooth device or host should be treated as one single entity in the current Bluetooth standard. The reason is that the modern smart device like smartphone involves multiple parties and could play a role of the platform for the installed apps which could share all Bluetooth resources. Also, we believe the profile-level authentication is necessary, and some kinds of standard verification procedure should be added.

Moreover, the device name and displayed icon is usually the only indicator for end users to distinguish the device. As a result, users may naively trust a device based on such indicator when doing pairing or connection operations. However, neither the device name nor the icon is reliable. The OS usually shows the icon just based on the claimed device type no matter what profiles it contains. Therefore, we think there should be a better mechanism to help users to verify the device.

IX. Related Work

In this section, we review the previous studies about the security issues on Bluetooth and peripheral devices.

Bluetooth Security. The early works on Bluetooth security focused on the vulnerabilities underlying the protocols and implementations. The early versions of Bluetooth have been found to suffer from attacks like sniffing [40], man-in-the-middle attack [31], PIN cracking [38], etc. On the Android platform, Naveed et al. [34] discovered the security issue of external device mis-bonding. The issue could enable an

unauthorized app downloading sensitive user data from an Android device. The similar vulnerabilities also exist on the iOS platform [21]. Different from them, our attacks aim to break Android system by abusing various Bluetooth profiles. BlueBorne [19] is an attack vector discovered in 2017 which contains 8 zero-day Bluetooth vulnerabilities across multiple platforms. This attack could penetrate and take control over targeted devices, even without pairing to the attacker's device. While, our attacks don't rely on any software bugs.

Some recent research concentrated on the security of Bluetooth Low Energy (BLE). For exmaple, Kolias et al. [29] indicated that BLE Beacon devices are susceptible to a variety of attacks, including Beacon hijacking, user profiling, presence inference, etc. Sivakumaran et al. [39] found some BLE devices allow unauthenticated reads and writes from third party devices. Also, Sławomir et al. [28] demonstrated several possible attacks on the GATT (Generic Attribute Profile) layer of the Bluetooth stack, and Ryan et al. [36] presented several techniques for eavesdropping BLE conversations.

In addition, some research has demonstrated the feasibility of user tracking exploiting Bluetooth. Das et al. [23] found majority of the fitness trackers use unchanged BLE address while advertising, making it feasible to track users. Korolova et al. [30] achieved cross-app user tracking through advertising packets broadcasted by nearby BLE-enabled devices. As a defense, Fawaz et al. [24] proposed a new device-agnostic defense system, called BLE-Guardian, that protects the privacy (device's existence) of the users/environments with BLE devices/IoTs.

In this paper, we target the latest Bluetooth stack and discover several high level *design* weaknesses which could lead to attacks causing severe consequences. These design flaws are not bounded to a specific platform or OS version.

Peripheral Devices and Security. In addition to Bluetooth, previous works also reveal that many other peripheral devices can be exploited to attack their host computers easily, with USB peripherals gaining the most attention. Wang et al. [48] introduced attacks targeting the physical USB connectivity between a smartphone and their computers. Maskiewicz et al. [33] presented a case study of the Logitech G600 mouse, demonstrating the feasibility of attacking airgapped peripherals. More recently, Su et al. [42] exploited the electrical properties of USB hubs and achieved crosstalk leakage attacks.

In 2014, Nohl et al. [35] proposed a comprehensive USB attack vector named BadUSB. They showed by registering a BadUSB device with multiple device types, it is possible to take any action on the host without authorization. To address this issue, Tian et al. [47] presented a defense solution named GoodUSB. They designed a permission model and a mediator to manage the risks during the enumeration phase of the USB protocol. This model is based on the insight that a device's identity should rely on the end user's expectation of the device's functionality. A series of research on the attack and defense of USB peripherals are conducted following this direction, including USBFILTER – a packet-level USB firewall [45], and ProvUSB – an architecture for block-level provenance for USB storage devices [44]. Also, Angel et al. [20] proposed a virtualization-based solution, which attaches peripheral devices to a logically separate, untrusted machine. Recently, Tian et al. [46] carried out a comprehensive survey on the research in USB security. The study suggests most of the USB attacks abuse the trust-by-default nature of the USB ecosystem and only a comprehensive defense solution expanding multiple layers would success in practice. We believe many issues residing in USB ecosystem also exist in the Bluetooth ecosystem in similar fashions, and the research in the Bluetooth domain could leverage the outcomes from the USB domain.

X. Conclusion

Bluetooth is an essential technique for short-distance and low-power communications and becomes more popular with the advent of the Internet of Things. The security of Bluetooth devices plays a critical role in protecting user's privacy and even personal safety. In this paper, we performed a systematic study over the Bluetooth profiles and discovered five design weaknesses. We further presented a series of attacks to demonstrate the feasibility and potential damages of such flaws on Android, including stealing information, app controlling, network sniffing, voice command injection, etc. Besides, we designed a defense solution on Android to effectively prevent such attacks. Moreover, we believe these newly discovered flaws are not just limited to a specific OS version. Broad Android versions are vulnerable, from 5.1 to the latest 8.1, and similar problems may also appear on other OS platforms. These flaws are rooted from the widely incorrect understandings and assumptions on the Bluetooth stack. We believe they should be just the tip of the iceberg, and the Bluetooth standard still needs a thorough security review.

Acknowledgement

We thank anonymous reviewers for their insightful comments. This work was partially supported by National Natural Science Foundation of China (Grant No. 61572415), and the General Research Funds (Project No. 14208818 and 14217816) established under the University Grant Committee of the Hong Kong Special Administrative Region, China. Wenrui Diao was supported in part by the Fundamental Research Funds for the Central Universities (No. 21618330).

References

[1] "Android Hidden API project on GitHub," https://github.com/anggrayudi/android-hidden-api, Accessed: May 2018.

[2] "AOSP Bluetooth Services," https://source.android.com/devices/bluetooth/services, Accessed: May 2018.

[3] "AOSP Input Overview," https://source.android.com/devices/input/, Accessed: May 2018.

[4] "AOSP Keyboard Device," https://source.android.com/devices/input/keyboard-devices, Accessed: May 2018.

[5] "Bluetooth Baseband Assigned Number," https://www.bluetooth.com/specifications/assigned-numbers/baseband, Accessed: May 2018.

[6] "Bluetooth Core Specification," https://www.bluetooth.com/specifications/bluetooth-core-specification, Accessed: May 2018.

[7] "Bluetooth-enabled Devices Worldwide in 2012 and 2018," https://www.statista.com/statistics/283638/installed-base-forecast-bluetooth-enabled-devices-2012-2018/, Accessed: May 2018.

[8] "Bluetooth Headset Profile," https://www.bluetooth.org/DocMan/handlers/DownloadDoc.ashx?doc_id=158743&_ga=2.24560186.1866324813.1527819756-910667369.1527819756, Accessed: May 2018.

[9] "Bluetooth Product on TI," http://www.ti.com/wireless-connectivity/simplelink-solutions/bluetooth-low-energy/products.html, Accessed: May 2018.

[10] "Bluetooth Profiles Overview," https://www.bluetooth.com/specifications/profiles-overview, Accessed: May 2018.

[11] "Bluetooth related CVEs," https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=bluetooth, Accessed: May 2018.

[12] "Bluez - Official Linux Bluetooth Protocol Stack ," https://www.bluetooth.com/specifications, Accessed: May 2018.

[13] "Dnsmasq Homepage," http://www.thekelleys.org.uk/dnsmasq/doc.html, Accessed: May 2018.

[14] "Programmatically pair Bluetooth device without the user entering pin - Stack Overflow," https://stackoverflow.com/questions/19047995/programmatically-pair-bluetooth-device-without-the-user-entering-pin, Accessed: May 2018.

[15] "PulseAudio Homepage," https://www.freedesktop.org/wiki/Software/PulseAudio/, Accessed: May 2018.

[16] "Pybluez Homepage," https://pybluez.github.io/, Accessed: May 2018.

[17] "Source Package on Debian - ofono," https://packages.debian.org/source/sid/ofono, Accessed: May 2018.

[18] "TCPDUMP/LIBPCAP Public Repository," https://www.tcpdump.org/, Accessed: May 2018.

[19] "The Attack Vector "BlueBorne" Exposes Almost Every Connected Device," https://www.armis.com/blueborne/, Accessed: May 2018.

[20] S. Angel, R. S. Wahby, M. Howald, J. B. Leners, M. Spilo, Z. Sun, A. J. Blumberg, and M. Walfish, "Defending against Malicious Peripherals with Cinch," in *Proceedings of the 25th USENIX Security Symposium (USENIX-SEC), Austin, TX, USA, August 10-12, 2016.*, 2016.

[21] X. Bai, L. Xing, N. Zhang, X. Wang, X. Liao, T. Li, and S. Hu, "Staying secure and unprepared: Understanding and mitigating the security risks of apple zeroconf," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland), San Jose, CA, USA, May 22-26, 2016*, 2016.

[22] A. Bellissimo, J. Burgess, and K. Fu, "Secure software updates: Disappointments and new challenges," in *1st USENIX Workshop on Hot Topics in Security, HotSec'06, Vancouver, BC, Canada, July 31, 2006*, 2006.

[23] A. K. Das, P. H. Pathak, C. Chuah, and P. Mohapatra, "Uncovering Privacy Leakage in BLE Network Traffic of Wearable Fitness Trackers," in *Proceedings of the 17th International Workshop on Mobile Computing Systems and Applications (HotMobile), St. Augustine, FL, USA, February 23-24, 2016*, 2016.

[24] K. Fawaz, K. Kim, and K. G. Shin, "Protecting Privacy of BLE Device Users," in *Proceedings of the 25th USENIX Security Symposium (USENIX-SEC), Austin, TX, USA, August 10-12, 2016*, 2016.

[25] Y. Fratantonio, C. Qian, S. Chung, and W. Lee, "Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop," in *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2017.

[26] J. Greig, "How Android P plans to turn your phone into a bluetooth keyboard or mouse," https://www.techrepublic.com/article/how-android-p-plans-to-turn-your-phone-into-a-bluetooth-keyboard-or-mouse/.

[27] T. Instruments, "Over-the-air download users guide for ble-stacktm version: 2.2.1," https://e2e.ti.com/cfs-file/__key/communityserver-discussions-components-files/538/CC2640-BLE-OAD-User_2700_s-Guide.pdf, October 2016.

[28] S. Jasek, "GATTacking Bluetooth Smart Devicess," in *Black Hat USA Conference*, 2016.

[29] C. Kolias, L. Copi, F. Zhang, and A. Stavrou, "Breaking BLE Beacons For Fun But Mostly Profit," in *Proceedings of the 10th European Workshop on Systems Security (EUROSEC), Belgrade, Serbia, April 23, 2017*, 2017.

[30] A. Korolova and V. Sharma, "Cross-App Tracking via Nearby Bluetooth Low Energy Devices," in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy (CODASPY), Tempe, AZ, USA, March 19-21, 2018*, 2018.

[31] D. Kügler, ""Man in the Middle" Attacks on Bluetooth," in *Financial Cryptography, 7th International Conference, FC 2003, Guadeloupe, French West Indies, January 27-30, 2003, Revised Papers*, 2003.

[32] S. Labs, "An1045: Bluetooth over-the-air device firmware update for efr32xg1 and bgm11x series products," https://www.silabs.com/documents/login/application-notes/an1045-bt-ota-dfu.pdf.

[33] J. Maskiewicz, B. Ellis, J. Mouradian, and H. Shacham, "Mouse Trap: Exploiting Firmware Updates in USB Peripherals," in *Proceedings of the 8th USENIX Workshop on Offensive Technologies, WOOT '14, San Diego, CA, USA, August 19, 2014.*, 2014.

[34] M. Naveed, X. Zhou, S. Demetriou, X. Wang, and C. A. Gunter, "Inside Job: Understanding and Mitigating the Threat of External Device Mis-Binding on Android," in *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS), San Diego, California, USA, February 23-26, 2014*, 2014.

[35] K. Nohl and J. Lell, "BadUSB–On accessories that turn evil," *Black Hat USA*, 2014.

[36] M. Ryan, "Bluetooth: With Low Energy Comes Low Security," in *Proceedings of the 7th USENIX Workshop on Offensive Technologies (WOOT), Washington, D.C., USA, August 13, 2013*, 2013.

[37] N. Semeconductor, "Updating firmware over the air," http://infocenter.nordicsemi.com/index.jsp?topic=%2Fcom.nordic.infocenter.tools%2Fdita%2Ftools%2FnRF_Connect%2FnRF_Connect_DFU.html, August 2018.

[38] Y. Shaked and A. Wool, "Cracking the Bluetooth PIN," in *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services (MobiSys), Seattle, Washington, USA, June 6-8, 2005*, 2005.

[39] P. Sivakumaran and J. B. Alís, "A Low Energy Profile: Analysing Characteristic Security on BLE Peripherals," in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy (CODASPY), Tempe, AZ, USA, March 19-21, 2018*, 2018.

[40] D. Spill and A. Bittau, "BlueSniff: Eve Meets Alice and Bluetooth," in *Proceedings of the First USENIX Workshop on Offensive Technologies (WOOT), Boston, MA, USA, August 6, 2007*, 2007.

[41] P. Stone, "Consider blocklisting qualcomm csr firmware update service," https://github.com/WebBluetoothCG/registries/issues/20, March 2017.

[42] Y. Su, D. Genkin, D. C. Ranasinghe, and Y. Yarom, "USB Snooping Made Easy: Crosstalk Leakage Attacks on USB Hubs," in *Proceedings of the 26th USENIX Security Symposium (USENIX-SEC), Vancouver, BC, Canada, August 16-18, 2017.*, 2017.

[43] C. Support, "Cs-327746-rp-1-training and tutorials - csr over-the-air-update," https://www.csrsupport.com/download/49800/CS-327746-RP-1-Training%20and%20Tutorials%20-%20CSR%20Over-the-Air-Update.pdf, March 2017.

[44] D. J. Tian, A. M. Bates, K. R. B. Butler, and R. Rangaswami, "Provusb: Block-level provenance-based data protection for USB storage devices," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS), Vienna, Austria, October 24-28, 2016*, 2016.

[45] D. J. Tian, N. Scaife, A. M. Bates, K. R. B. Butler, and P. Traynor, "Making USB Great Again with USBFILTER," in *Proceedings of the 25th USENIX Security Symposium (USENIX-SEC), Austin, TX, USA, August 10-12, 2016.*, 2016.

[46] D. J. Tian, N. Scaife, D. Kumar, M. Bailey, A. Bates, and K. Butler, "SoK: "Plug & Pray" Today – Understanding USB Insecurity in Versions 1 through C," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland), San Francisco, CA, USA, May 21-23, 2018*, 2018.

[47] J. D. Tian, A. M. Bates, and K. R. B. Butler, "Defending Against Malicious USB Firmware with GoodUSB," in *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC), Los Angeles, CA, USA, December 7-11, 2015*, 2015.

[48] Z. Wang and A. Stavrou, "Exploiting Smart-Phone USB Connectivity For Fun And Profit," in *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC), Austin, Texas, USA, 6-10 December 2010*, 2010.

[49] C. Xiao, "Update: Xcodeghost attacker can phish passwords and open urls through infected apps," http://researchcenter.paloaltonetworks.com/2015/09/update-xcodeghost-attacker-can-phish-passwords-and-open-urls-though-infected-apps/, September 2015.