# BLAP: Bluetooth Link Key Extraction and Page Blocking Attacks

Changseok Koh
*Korea University*
mrchangsuck@gmail.com

Jonghoon Kwon
*ETH Zürich*
jong.kwon@inf.ethz.ch

Junbeom Hur
*Korea University*
jbhur@korea.ac.kr

***Abstract*—Secure Simple Pairing (SSP) and Link Manager Protocol (LMP) authentication are two main authentication mechanisms in Bluetooth specification. In this paper, we present two novel attacks, called *link key extraction* and *page blocking attacks*, breaking LMP authentication and SSP authentication, respectively. The link key extraction attack allows attackers to extract link keys of Bluetooth devices generated during the SSP procedure by exploiting Bluetooth HCI dump. The page blocking attacks by man-in-the-middle (MITM) attackers enforce Bluetooth connections, enabling subsequent SSP downgrade attacks to bypass the SSP authentication challenge. In order to demonstrate the efficacy, we implement our attacks on various real-world devices and show that (1) a target link key is dumped into a log and extracted efficiently, possibly leading to the subsequent impersonation attack, and (2) malicious MITM connections can be established with 100% success rate, enabling subsequent SSP downgrade attack. We investigate the root causes for the vulnerabilities and present mitigations.**

## I. Introduction

Authentication mechanisms of Bluetooth, such as pairing and Link Manager Protocol (LMP) authentication, are the first-line defense for billions of users to protect their devices from unauthorized pairing attempts. To authenticate a newly connected device, the pairing process generates a shared secret, called a link key. The link key is the only hidden value of security parameters for LMP authentication and encryption key generation. For the quick resumption of future sessions, the link key can be saved after the initial pairing and reused, omitting the later pairing procedures. LMP authentication enables the paired devices to authenticate each other by challenging whether they possess the same link key. In order to improve the pairing experiences and security, the Bluetooth standard introduced *Secure Simple Pairing (SSP)* since Bluetooth v2.1. Specifically, SSP has the following four authentication mechanisms with different security levels: (1) Just Works, (2) Numeric Comparison, (3) Out of Band, and (4) Passkey Entry. Furthermore, SSP—except Just Works—provides resilience against man-in-the-middle (MITM) attacks by taking advantage of public-key cryptography.

Despite the authentication mechanisms, with its growing popularity, Bluetooth has inevitably emerged as a major attack surface; the specification flaws of SSP [1]–[6] and the implementation flaws in Android, iOS, Windows, and Linux made by each vendor [7]–[10] have been exploited. BIAS [7] presented how to perform impersonation attacks by breaking the LMP authentication. KNOB [8] presented an attack method on the subsequent encryption key negotiation protocol by manipulating the Bluetooth firmware layer. In addition, downgrading SSP has been discussed in several previous studies. The attack methods posed a question "*what security guarantees do the Bluetooth authentication mechanisms actually provide?*".

In this paper, we present and demonstrate two novel attacks for **persistent Bluetooth impersonation**, exploiting the Bluetooth authentication mechanisms with increased viability and effectiveness: *link key extraction attack* and *page blocking attack*. The proposed attacks aim to break LMP authentication and pairing authentication, respectively, considering the target device's status, either bonded or non-bonded.

First, the link key extraction attack extracts link keys of Bluetooth devices by exploiting the Bluetooth protocol log (i.e., *HCI dump*) in which link keys are logged. The attack exhibits a significant impact on Bluetooth security compared to the previous attacks: 1) the attack aims to extract semi-permanent (saved) link keys. Once the key is extracted, the attacker can continuously exploit it for multiple sessions, breaking forward secrecy. Note that many existing attacks are only valid for a single session. 2) The attack requires access to the Bluetooth protocol layer that is typically open to users, such that it increases practicality, while the previous attacks require firmware layer manipulation.

Second, the page blocking attack establishes a MITM connection in a deterministic manner, which can be subsequently leveraged to downgrade SSP to Just Works mode. Although downgrading SSP has been discussed in several previous studies [1]–[3], how to establish and implement the MITM attack on Bluetooth connections have never been discussed in the literature. Since the Bluetooth handshake transcripts for connection establishment are delivered through a wireless channel, it is practically challenging to force the victim device to justly connect to the attacker's device in practice[1]. Our page blocking attack solves this problem by exploiting the novel vulnerability we found; the lack of verification procedure of the Bluetooth authentication mechanism that checks whether the connection initiator actually initiates the pairing.

The main contributions of our work are summarized as:

- We first present a link key extraction attack that exploits the security flaw in the HCI dump, which records all data passed through the HCI interface in a log file, including link keys. We then describe how to extract link keys by exploiting the HCI flaw against Android and Windows systems in practice.
- Next, we propose a page blocking attack that ensures MITM connection establishment between a victim and

[1]On the basis of our experiment, the success rate of establishing the MITM connection shows 42∼60%. More detailed experiment results will be explained in Section VI.

attacker devices in a deterministic manner. It then leads to a subsequent SSP downgrade attack to Just Works by exploiting the lack of the authentication mechanism for the connection initiator.

- We demonstrate the efficacy of the link key extraction and page blocking attacks via real-world implementations. More precisely, our implementation of the link key extraction shows that various systems (i.e., six smartphones and two PC systems) are logging 128-bit link keys into the HCI dump, which are extractable. We also implement the page blocking attack on seven smartphones, and show that the attack can force the victim devices to be connected to the attacker's device with a 100% success rate and then downgraded to Just Works.
- To thwart our discovered attacks, we propose several short-term mitigation mechanisms that are deployable in the current Bluetooth networks and also discuss potential long-term mitigation strategies.

**Responsible Disclosure.** We reported our findings and technical details to the Bluetooth SIG, and claimed the necessity for the specification revision such as the link key payload encryption. We are now waiting for their responses, as of April 5th, 2022.

## II. Background

Bluetooth technology is implemented based on either Basic Rate Enhanced Data Rate (BR/EDR) or Bluetooth Low Energy (BLE). BR/EDR, also known as classic Bluetooth, usually suits for high throughput services such as hands-free, audio distribution that are normally used in car infotainment and headset devices. Whereas, BLE, also known as Bluetooth Smart, is designed to support applications that have lower throughput and duty cycle such as IoT devices. The two variants of the Bluetooth technology share a core system architecture in common, which consists of three components: host stack (or host), controller, and host controller interface (HCI). Above the core system, Bluetooth application services can be run at the application layer. Fig. 1 describes a typical architecture of Bluetooth system. Server/client interoperability of application services is accomplished by the specifications called Bluetooth profile such as Hands-Free profile (HFP), Phone Book Access Profile (PBAP), and the network functionalities for the profiles are supported by the core components.

### A. Bluetooth Architecture

The Bluetooth core components are described as follows.

*1) Host Stack:* Host stack runs core protocols to manage connection and logical links for profile services in the application layer. The core protocols include Generic Access Profile (GAP) for device connection, Service Discovery Protocol (SDP) for service discovery and connection, and Logical Link Control and Adaptation Protocol (L2CAP) for data fragmentation, reconstruction, and per-channel communications, for instance. Some host stack solutions are implemented based on open source projects such as BlueZ [11] and Bluedroid [12], and can be customized without much difficulty in practice.

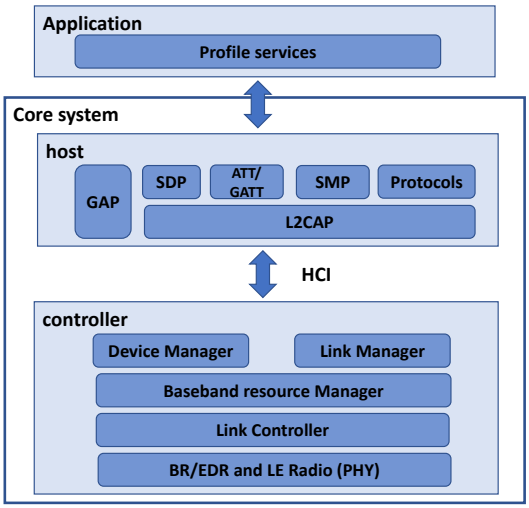



Fig. 1: Bluetooth system architecture

*2) Controller:* The controller manages the traffic of Bluetooth communications between devices. Link Manager Protocol (LMP) in the controller supports security operations, including key generation, device authentication, and data encryption and decryption. Its software implementation is typically dependent on the hardware structures within chipset. For this reason, it is almost infeasible for the third party developers to install their customized operations to the controller.

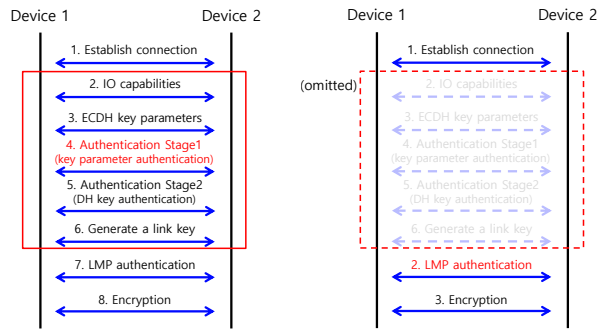
*3) HCI:* It enables data communication between the host and controller via a serial interface. HCI is defined in the Bluetooth specification [13] as a set of commands and events for the host and the controller to interact with each other, along with a data packet format and a set of rules for flow control. Since HCI is separated from the host, it allows a host stack to be deployed with one or more controllers independently. Furthermore, HCI is used to deploy a protocol tracking tool, called *HCI dump*, which records whole HCI data into log files.

### B. Bluetooth BR/EDR Discovery and Connection

BR/EDR devices can discover each other and establish a new connection with the target device by the following procedures. Any device can be either an initiator or a responder.

*1) Target Discovery:* A responder device is set to a discoverable mode, and wait for receiving any inquiry message. An initiator device broadcasts inquiry message to every device in the supported signal range and requests a response message from any responder. When any potential responder device receives the message, it responds to the inquiry by broadcasting its information such as Bluetooth address (BDADDR), device name, and supporting service.

*2) Connection Establishment:* The initiator device begins the connection establishment by sending a *page* request to the BDADDR of a discovered target device. The target responder then sends a page response message to the initiator. Finally, the initiator assigns an LT_ADDR, which is a logical transport address, to the responder to establish a Bluetooth connection.




(a) For non-bonded devices (b) For bonded devices

Fig. 2: Pairing and authentication procedures

To prevent unintentional connections, a responder may set the non-connectable mode to disable the page procedure.

### C. Bluetooth Pairing and Authentication

Once two devices are connected, they conduct pairing, LMP authentication and encryption to enforce the security rules as shown in Fig. 2.

*1) Secure Simple Pairing:* Pairing refers to a procedure of authenticating a newly connected device. In the legacy pairing, a PIN number is manually entered. However, as it has been recognized as vulnerable to diverse attacks [14], [15], Secure Simple Pairing (SSP) was developed as a cryptographically-enforced protocol. If two devices are connected, they exchange their IO capabilities[2] and ECDH key parameters. They then authenticate each other based on the ECDH public parameters and secret values exchanged in the previous step in the 'Authentication Stage 1'; and finally the same link key is derived in each device from the secret key (called DHKey) agreed using ECDH algorithm as shown in Fig. 2a.

Once a link key is generated, both devices may save it to reuse for subsequent authentication (called 'bonding') such that the later pairing procedures can be omitted for the 'bonded' devices. For bonded devices, only LMP authentication is performed based on the previously shared link key as shown in Fig. 2b.

*2) LMP Authentication and Encryption:* For the bonded devices, SSP is omitted and only the LMP authentication is launched between them. During the LMP authentication procedure, each device checks if they possess the same link key via a challenge-response protocol. Specifically, a verifier challenges a prover with two arbitrary values, and the prover then calculates and responds with a hash of the values and its link key. On receipt of it, the verifier can confirm whether the prover possesses the same link key by comparing its local calculation result and the response value.

[2]Depending on the selected IO capabilities, the suitable association model is selected among the following four options: Numeric Comparison, Just Works, Out of Band, and Passkey Entry. For example, Numeric comparison is launched when both devices have capabilities of displaying a six-digit number and providing a 'Yes/No' input method for manual comparison and authentication code confirmation.

After the LMP authentication, both devices generate an encryption key using an encryption key generator that takes the following parameters: a public random number, an extra result value obtained from the challenge-response procedure in LMP authentication, and a link key.

## III. System and Attack Models

### A. System Model

We suppose a Bluetooth system composed of three devices, $\mathcal{A}$, $\mathcal{M}$, and $\mathcal{C}$, each of which plays different roles as follows. $\mathcal{M}$ is a device with sensitive data which can be shared via Bluetooth profile services such as Phone Book Access Profile (PBAP), Hands-Free Profile (PBAP), and Message Access Profile (MAP) (e.g., mobile phone). $\mathcal{C}$ is another device trying to pair with $\mathcal{M}$ as a trusted client (e.g., car-kits, headset devices, or PC). $\mathcal{A}$ is a malicious device aiming to connect to the Bluetooth services in $\mathcal{M}$ to access sensitive data by impersonating $\mathcal{C}$. SSP authentication and LMP authentication procedures are supposed to protect the Bluetooth communications between $\mathcal{M}$ and $\mathcal{C}$ for each bonded or non-bonded case. However, we will demonstrate how our two attacks break them in the subsequent sections.

### B. Attack Model and Assumption

In our attack model, an attacker's final goal is to Bluetooth connect to $\mathcal{M}$ in order to mine sensitive information. To this end, the attacker establishes a long-term presence around $\mathcal{M}$ (i.e., a hard target), collecting a list of pairing devices (i.e., soft targets) that are easily accessible and relatively unprotected. Once $\mathcal{C}$ is determined, the attacker harvests pairing information of $\mathcal{C}$, impersonates $\mathcal{C}$ using $\mathcal{A}$, and establishes an illicit and *persistent* connection to $\mathcal{M}$. In the attack, we assume that the attacker $\mathcal{A}$ can (1) silently access and manipulate $\mathcal{C}$, (2) extract HCI dump from $\mathcal{C}$, (3) sniff USB data of HCI from $\mathcal{C}$, and (4) obtain BDADDR of $\mathcal{C}$.

As for the first assumption, there are many real-world scenarios where $\mathcal{C}$ (e.g., car-kit, headset device, etc) receives less security attention than $\mathcal{M}$ (e.g., smartphones). For instance, on-board infotainment devices do not directly store secure information. They thus tend to be easily shared without much security concern, making $\mathcal{C}$ vulnerable to physical access by $\mathcal{A}$. As for the second assumption, some operating systems provide HCI dump tools. For example, Android supports a background HCI dump option among its native hidden menu, called 'Bluetooth HCI snoop log' that anyone can activate in a straightforward way. As for the third assumption, there are various USB analyzers such as 'Free USB Analyzer' [16], which are free to use. Thus, it is also not a strong assumption that the attacker can sniff the USB interface for HCI data extraction in practice. Finally, for BDADDR tracking, Cominelli et al. [17] proved that it is possible to calculate an actual BDADDR from Bluetooth signals in 4 seconds.

Therefore, we consider that our attack model and assumptions are reasonable in practice, posing realistic threats. On

the basis of the attack models and assumptions, we describe the details of attack implementations in the next sections.

## IV. Link Key Extraction Attack

Link key extraction attack aims to extract bonded link keys. Once a link key is extracted, $\mathcal{A}$ can leverage it continuously for impersonation attacks as well as eavesdropping against $\mathcal{M}$. Thereby sensitive Bluetooth data such as phone books, messages, and phone call conversations of $\mathcal{M}$ will be continuously leaked.

Although link keys are used by the controller, it typically has limited storage for only supporting light, small, and low-powered Bluetooth chipset designs. Thus, the host stack which typically has enough storage is used to manage (e.g., storing and reloading) link keys. Whenever a link key is generated, the controller sends the link key to the host stack through HCI via HCI_Link_Key_Notification for the purpose of future reuse. After that, on the subsequent re-connection, if LMP needs the link key for a certain device, the controller requests the link key to the host using HCI_Link_Key_Request, and then the host replies to the event with a corresponding key via HCI. While a link key is loaded in the messages it is not protected, i.e. the link key is transmitted through the HCI as plaintext. Furthermore, the HCI may leak its data more easily than the host and the controller. Thus, the attacker extracts HCI data of a target device for the purpose of obtaining a target link key, which is the point that our link key extraction attack exploits. Next, we show how to extract HCI data, and describe our attack procedure.

### A. Extracting HCI Data from HCI Dump

The HCI dump is a widely used HCI logging method by Bluetooth implementations [18], [19]. It allows users to log the whole HCI data in RFC 1761 format [20] between a host and a connected controller. The link key related HCI messages, such as HCI_Link_Key_Request and HCI via HCI_Link_Key_Notification, are also logged by the HCI dump, including a link key in their payload.

The HCI dump log can be easily parsed as shown in Fig. 3. In the figure, a mobile phone and a headset device are bonded with a link key '71bb87cecb...', and the key information is captured from the HCI in the mobile phone. When they start to authenticate each other, each controller requests a corresponding link key to the connected host with HCI_Link_Key_Request. The bottom of the figure shows that HCI_Link_Key_Request_Reply, which is the reply command for the event, includes the corresponding link key. If an attacker can get the HCI dump from a device, he can exploit it to extract link keys in the device.

Some Linux-based operating systems support SW tools of HCI dump. For example, one can launch an HCI dump after installing the 'bluez-hcidump' package on Ubuntu, which can be an effective attack surface for the attacker. Android OS also provides an HCI dump tool. However, no installation of the hcidump package is required, as the HCI dump module is built in the package of the Android host stack solution

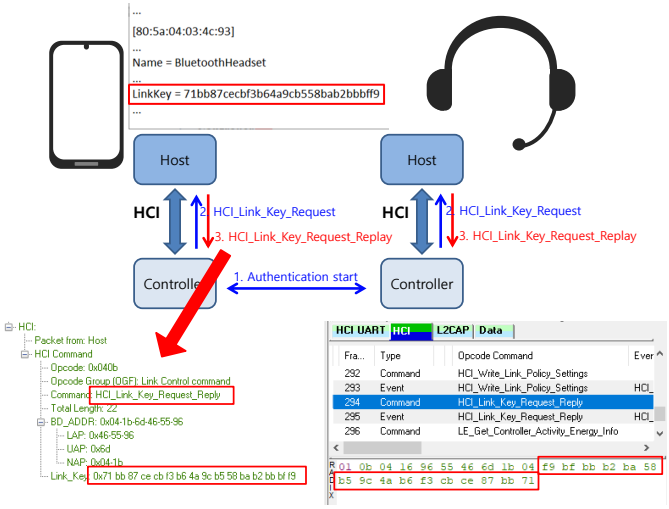



Fig. 3: A link key in a HCI packet and its HCI dump

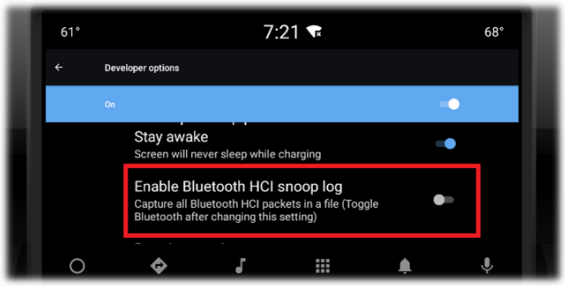



Fig. 4: HCI dump menu in Android Automotive platform

by default. Further, the module can be easily executed via Android's native menu, 'Android developer options', a set of hidden menu. Therefore, anyone can activate it by operating an Android settings app, tapping the build number several times [21], and performing the HCI dump via 'Bluetooth HCI snoop log' option. The snoop log option enables background logging for HCI dump, which can be extracted by users via 'Android bug report' [22] without any system access permission. Hence, it is straightforward for the attacker to extract an HCI dump log, including link keys, from an Android platform.

Capturing HCI dump from released products is also possible by leveraging their HW ports. Many vendors (e.g., Bluetooth headset manufacturers) provide HCI dump tools for debugging purposes. Although they require hardware access, such as debugging port wiring, it may not be difficult for well-motivated attackers to carry out the attack.

### B. Extracting HCI Data via USB Sniffing

While some platforms (e.g., Android OS and Linux-based OS) provide HCI dump, the others (e.g., Microsoft Windows host stack and CSR harmony host stack solutions) do not support it. However, the HCI data can still be leaked through its hardware interface.

There are typically two types of Bluetooth chipsets. One is the 'controller-type' consisting of controller components. It is

connected to an Application Processor which has host components inside. The other is the 'stand alone-type' which consists of the whole core components, including host stack, HCI, and controller. The Bluetooth system using the controller-type chipset employs physical transport interfaces such as UART or USB for the HCI. It is thus possible to sniff the hardware port with capturing equipment. For example, FTE commercially provides a device called FTS4USB™ [23] that can capture UART/USB HCI data by connecting electronic wires to hardware ports between AP and controller.

Windows OS can also support a USB-type Bluetooth dongle as the Bluetooth controller. Specifically, the host stack is implemented on CPU, and the host and the controller are connected via USB, which is the hardware interface of the HCI. To extract HCI data, the attacker may sniff the USB using various USB analyzers available online for free. In Section VI-B1, we describe how to extract a link key via USB sniffing in practice.

### C. Attack Procedure

Direct extraction of HCI data from a private device $\mathcal{M}$ would be difficult in a naive way because all of the preconditions, such as executing HCI dump or USB sniffing, can hardly be satisfied in practice. For example, when the attacker accesses $\mathcal{M}$ and tries to silently extract the HCI dump log, it may fail if $\mathcal{M}$ is locked. Hence, our attack model does not aim to extract a link key directly from the victim device. Rather, by taking advantage of the fact that a pair of bonded devices share exactly the same link key, the attacker aims to leverage the link key from $\mathcal{C}$ which can be easily shared with the attacker (e.g., headset, car-kit, or PC), and deliver the attack against $\mathcal{M}$.

In the attack procedure, the attacker can sniff physical interface of HCI on Windows PC, or gather HCI dump logs on Android systems such as an Android automotive platform. Fig. 4 shows the Android automotive platform providing HCI dump functionality from the developer options menu. (The real-world implementation of HCI data extraction is described in Section VI-B1.)

As shown in Fig. 5, the link key extraction attack progresses as follows:

1) $\mathcal{A}$ accesses $\mathcal{C}$ and manipulates it to record HCI data such as HCI dump or USB sniffing.
2) As $\mathcal{A}$ targets to extract the bonded link key from $\mathcal{C}$, he changes BDADDR of $\mathcal{A}$ to impersonate $\mathcal{M}$
3) $\mathcal{C}$ establishes a connection and initiates LMP authentication procedure with $\mathcal{A}$. As $\mathcal{A}$ spoofs $\mathcal{M}$, the controller in $\mathcal{C}$ requests to its host the link key associated with $\mathcal{M}$.
4) The host responses to the link key request from its controller, at this time, the link key is logged into a HCI dump log.
5) The attacker disconnects the link at the beginning of the LMP authentication.
6) The attacker extracts HCI dump, and finally gathers the link key between $\mathcal{C}$ and $\mathcal{M}$ by analyzing the HCI dump.

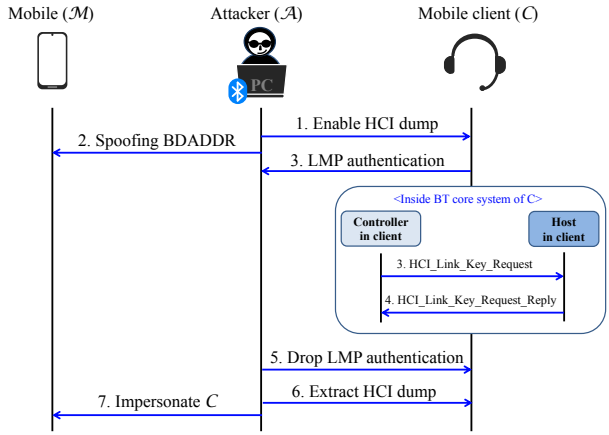



Fig. 5: Link key extraction attack procedure

7) By leveraging the link key extracted from $\mathcal{C}$, $\mathcal{A}$ can mount impersonation attack later against $\mathcal{M}$ and finally extract its private data.

In step 3), any authentication failure may cause the elimination of the link key in the device of $\mathcal{C}$, leading to the failure of LMP authentication of our attack. Since controllers typically request the link keys from their hosts before executing the LMP authentication, having $\mathcal{C}$ be an authentication initiator (in step 3) and dropping LMP authentication by $\mathcal{A}$ (in step 5) ensures recording of the link key in HCI dump (in step 4) without authentication failure. The link key extracted from $\mathcal{C}$ can also be used by $\mathcal{A}$ for an impersonation attack against $\mathcal{M}$. Additionally, $\mathcal{A}$ would be able to decrypt not only the future, but also the past communications of $\mathcal{M}$ captured by air-sniffers using the key.

## V. Page Blocking Attack

In this section, we first describe previous SSP downgrade attack and its practical limitation. We then describe our page blocking attack and discuss its practical implication.

### A. Previous SSP Downgrade Attacks and Limitation

SSP adopts ECDH key exchange algorithm for secure communications. However, among the four SSP authentication modes, Just Works mechanism cannot verify the authenticity of the exchanged ECDH key parameters, since it is designed for devices without IO capability such as a headset device. Thus, Just Works is not immune to MITM attacks and many previous MITM attacks on SSP [3], [24], [2], [1] have focused on downgrading the other more secure modes to Just Works to avoid manual verification of the ECDH parameters. Downgrade attack can begin by setting a spoofing device to 'NoInputNoOutput' IO capability. When two devices start to run the SSP protocol, they first exchange their IO capability information to determine which of the SSP association modes shall be executed. If at least one of them is set as NoInputNoOutput mode, due to the lack of IO capability, Just Works automatically confirms the authenticity of exchanged ECDH

key parameters without user verification and the spoofing device can pass SSP authentication challenges.

The downgrade attack seems straightforward in a theoretical aspect, but they just assumed that $\mathcal{M}$ and $\mathcal{A}$ have been already connected before the attack by any means. However, in practice, when (potential victim) $\mathcal{M}$ tries to initiate a pairing procedure with $\mathcal{C}$, $\mathcal{M}$ would connect to one of $\mathcal{C}$ and $\mathcal{A}$ in an indeterministic way, implying the attacker cannot guarantee $\mathcal{M}$ will be actually connected with the malicious device $\mathcal{A}$. Contrarily, our page blocking attack can enforce such malicious connections to the attacker's device $\mathcal{A}$ in a deterministic manner.

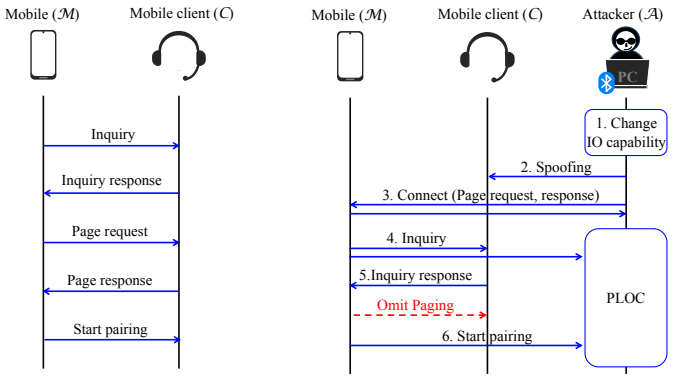
Besides, once a Bluetooth network is established, the BDADDR of each device is no longer used. The connection initiator assigns the responder an LT_ADDR which is a logical transport channel in the Bluetooth network to address a message destination on subsequent data communications. For this reason, the same BDADDR between $\mathcal{C}$ and $\mathcal{A}$ is valid only during the initial stage of Bluetooth network establishment, of which time window size is very short in practice. Thus, if $\mathcal{C}$ responds to the page request earlier than $\mathcal{A}$, $\mathcal{C}$ is more likely to be connected to $\mathcal{M}$, which might reduce the possibility of MITM establishment of $\mathcal{A}$.

To overcome this limitation, our page blocking attack makes $\mathcal{A}$ work as the connection initiator while making $\mathcal{M}$ still seem to be the connection initiator from the victim's point of view. After that, our attack allows subsequent SSP downgrade attacks to be conducted without any signs of abnormality, since it works as usual by our design. Specifically, our attack exploits the lack of verification procedure of the Bluetooth authentication mechanism that checks whether the connection initiator actually initiates the pairing.

### B. Attack Procedure

Page blocking attack aims to make $\mathcal{A}$ definitely establish a MITM connection to $\mathcal{M}$ in the unreliable wireless communication environment. To this end, we design 'Physical Layer Only Connection' (PLOC). Detailed attack procedures are explained in two parts in this section.

*1) Page Blocking Attack Procedure:* Page blocking attack begins by making $\mathcal{A}$ initiate the connection instead of $\mathcal{M}$. However, subsequent pairing attempts of $\mathcal{A}$ may fail as it will suddenly show a pairing confirmation popup on the display of $\mathcal{M}$ at an unexpected time. Thus, in order to ensure that the popup shall not appear or at least be displayed as soon as the user has attempted to pair, we first confined the role of $\mathcal{A}$ to the connection initiator (rather than a responder) and $\mathcal{M}$ to the pairing initiator in page blocking attack. In the attack, we also make $\mathcal{A}$ stay connected without proceeding a host layer connection procedure (which is called 'PLOC') until $\mathcal{M}$ sends a pairing request to $\mathcal{A}$. While under PLOC with a spoofing device, whenever $\mathcal{M}$ initiates a pairing procedure with $\mathcal{C}$, the pairing request would be certainly sent to $\mathcal{A}$, as $\mathcal{M}$ considers it is already connected with $\mathcal{C}$ so that it will omit the subsequent connection procedure and send the pairing request directly to the established link which is actually connected to $\mathcal{A}$.




(a) Normal pairing (b) Pairing under page blocking attack

Fig. 6: Pairing procedures

Page blocking attack is a kind of man-in-the-middle attack on the connection establishment protocol between two devices. Fig. 6a describes a normal pairing procedure between $\mathcal{M}$ and $\mathcal{C}$. $\mathcal{M}$ first discovers nearby devices (using inquiry and response messages). Among the discovered devices, $\mathcal{M}$ establishes Bluetooth connection with $\mathcal{C}$ (using page request and response) if $\mathcal{C}$ is the selected responder, and then it initiates a pairing procedure with $\mathcal{C}$. Page blocking attack intervenes in the above protocol. Fig. 6b depicts the attack procedure which progresses as follows:

1) $\mathcal{A}$ changes its IO capability to NoInputNoOutput.
2) $\mathcal{A}$ impersonates $\mathcal{C}$ by spoofing the BDADDR.
3) $\mathcal{A}$ establishes a connection to $\mathcal{M}$ and stays in PLOC.
4) $\mathcal{M}$ broadcasts the inquiry message to discover nearby devices including $\mathcal{A}$ and $\mathcal{C}$.
5) Once $\mathcal{C}$ receives the inquiry message, it replies to $\mathcal{M}$.
6) $\mathcal{M}$ begins the pairing procedure. Since $\mathcal{M}$ is already connected with the spoofing device $\mathcal{A}$, the host of $\mathcal{M}$ omits the connection re-establishment procedure with $\mathcal{C}$, and initiates the pairing procedure with $\mathcal{A}$ through the connected link rather than $\mathcal{C}$ without perceiving it.

In the attack, as IO capability of $\mathcal{A}$ is set to NoInputNoOutput (step 1), the subsequent pairing will proceed in Just Works mode, thereby SSP downgrade is realized because the challenge from SSP authentication stage 1 will be bypassed. In addition, $\mathcal{M}$ can function as ordinary while staying in PLOC since most mobile devices are implemented for supporting multiple connections in practice. Thus, $\mathcal{M}$ performs device discovery as well as pairing steps (step 4 and 6) as above.

*2) Subsequent SSP Downgrade Attack:* When $\mathcal{M}$ initiates the pairing, whether it displays a confirmation popup or not depends on its Bluetooth version. Regarding the confirmation popup issue, the security guideline for SSP authentication stage 1 in the Bluetooth specification defines how to validate exchanged key parameters. Fig. 7 partially shows the mapping information for each Bluetooth version; 'DisplayYesNo' device denotes the capability of displaying a six-digit number and taking yes or no choice from users. 'NoInputNoOutput' indicates the device is not equipped with any IO interface. As

| | | Device A (Initiator) | |
|---|---|---|---|
| | | DisplayYesNo | NoInputNoOutput |
| Device B (Responder) | DisplayYesNo | Numeric Comparison: Both Display, Both Confirm. | Numeric Comparison with automatic confirmation on device A only. |
| | NoInputNoOutput | Numeric Comparison with automatic confirmation on device B only. | Numeric Comparison with automatic confirmation on both devices. |

(a) IO capability mapping on version 4.2 and lower

| | | Device A (Initiator) | |
|---|---|---|---|
| | | DisplayYesNo | NoInputNoOutput |
| Device B (Responder) | DisplayYesNo | Numeric Comparison: Both Display, Both Confirm. | Numeric Comparison with automatic confirmation on device A only and Yes/No confirmation whether to pair on device B. Device B does not show the confirmation value. |
| | NoInputNoOutput | Numeric Comparison with automatic confirmation on device B only and Yes/No confirmation on whether to pair on device A. Device A does not show the confirmation value. | Numeric Comparison with automatic confirmation on both devices. |

(b) IO capability mapping on version 5.0 and higher

Fig. 7: (Partially displayed) IO capability mapping for authentication stage 1

shown in Fig. 7, the validation depends on IO capabilities of the pairing initiator and responder. Specifically, when either the initiator or responder is a NoInputNoOutput device, Just Works shall be launched due to the automatic confirmation. In Bluetooth version 4.2 or lower, there is no mandated guideline for the confirmation popup, so most implementations automatically confirm the pairing without any user confirmation when working as the initiator. Whereas, when they are working as the responder, most implementations ask for users' confirmation through a notification to prevent silent pairing by Just Works associations. In version 5.0 or higher, displaying a confirmation popup is mandated on DisplayYesNo devices. However, the confirmation in every version only asks if users would accept the pairing or not, thus it is difficult for users to determine whether the pairing is actually being performed between legitimate devices.

Consequently, $\mathcal{A}$ can establish the connection and perform pairing with $\mathcal{M}$. During the Just Works pairing, $\mathcal{M}$ will accept the pairing automatically without any user confirmation in case of version 4.2 or lower. For version 5.0 or higher, $\mathcal{M}$ will require user confirmation for the pairing. Nonetheless, the victim user will highly likely accept the confirmation since: (1) the pairing was intended by the victim user, (2) popup is immediately displayed after the intended pairing initiation, and (3) there is no way to determine whether the pairing is actually conducted between $\mathcal{C}$ and $\mathcal{M}$.

## VI. Implementation & Evaluation

In this section, we describe how to implement link key extraction and page blocking attacks, and demonstrate their efficacy with real-world implementation.

```
[ ./include/bt_target.h]
@@ -464,7 +464,7 @@
#ifndef BTA_DM_COD
-#define BTA_DM_COD {0x5A, 0x02, 0x0C}
+#define BTA_DM_COD {0x3c, 0x04, 0x04}
 #endif
```

Fig. 8: Snippet for COD modification

```
[ ./stack/btu/btu_hcif.c]
@@ -226,7 +227,7 @@ void btu_hcif_process_event(...)
            btu_hcif_pin_code_request_evt (p);
            break;
        case HCI_LINK_KEY_REQUEST_EVT:
-           btu_hcif_link_key_request_evt (p);
+//         btu_hcif_link_key_request_evt (p);
            break;
```

Fig. 9: Snippet for ignoring HCI_Link_Key_Request event

### A. Experiment Setup

We use a Nexus 5x Android device as $\mathcal{A}$ in this evaluation. In order to install our attack implementation on it, we need to alter the host stack library (bluedroid) and BDADDR files. However, because they are stored in read-only file systems, we need to first remount the Android system partition. Remounting the system partition can be accomplished by flashing locally built boot.img to the device. To generate and flash a locally built image, we execute the following steps using Android open-source android-6.0.1_r8 [25] as the base code:

1) Flash an MMB29P Google factory image to Nexus 5x to avoid any boot failure caused by the mismatch between the OS version and locally built boot.img,
2) Download and build android-6.0.1_r8 source code as user-debug mode,
3) Unlock the OEM locking menu in Nexus 5x's developer options to enable flashing boot.img,
4) Launch the Android bootloader and flash locally built boot.img to the device, and
5) Lock the OEM locking state using the bootloader command 'fastboot oem locking', and then reboot the Android OS.

After flashing locally built boot.img to the device, we can get Android superuser privilege and then remount the system partition to writable mode. The host stack library (bluedroid) and BDADDR files can then be altered in the Nexus 5x. for spoofing a device, we need to acquire the attributes of the target device and write them to the Nexus 5x, such as BDADDR and Class Of Device (COD). For the BDADDR, Nexus 5x uses a persistent '/persist/bdaddr.txt' file to store the local BDADDR information in ASCII string format. Thus, we can simply change local BDADDR by modifying it. Likewise, the COD of Nexus 5x can be easily changed by simply modifying 'system/bt/in-clude/bt_target.h', which is a definition file of the host stack library, for the settings of the local Bluetooth system. Fig. 8 shows the process of changing COD from mobile device type (0x5A020C) to hands-free device type (0x3c0404). By building and pushing the host stack library 'bluetooth.default.so' into '/system/lib/hw/', we can alter the host stack in the Nexus 5x.

### B. Attack Implementation

*1) Link Key Extraction Attack:* We conduct the attack in two different environments. First, we implement the attack on various Android devices and leverage HCI dump to extract link keys from them. Second, we implement the attack on Windows 10 and Linux ubuntu 20.04 and extract HCI data from the physical interface of HCI, specifically via USB sniffing. The attacks are implemented based on the attack procedure given in Section IV-C. To allow $\mathcal{A}$ to disconnect the link at the beginning of LMP authentication (step 5), we modify the host stack as shown in Fig. 9; calling function btu_hcif_link_key_request_evt() required to process HCI_Link_Key_Request event is skipped, disconnecting the link due to timeout. Specifically, when $\mathcal{C}$ initiates the LMP authentication, HCI_Link_Key_Request events are delivered from the controllers to the hosts on both $\mathcal{C}$ and $\mathcal{A}$. At that time, our attack code makes $\mathcal{A}$ wait indeterminately, and thereby link would be disconnected by the timeout of the session, while the host in $\mathcal{C}$ replies with a corresponding link key to its controller. Therefore, the target link key would be logged by HCI dump in $\mathcal{C}$ and the connection will be disconnected without authentication failure which may cause the expiration of the existing link key on $\mathcal{C}$.

**Link key extraction via HCI dump.** We use Nexus 5x running Android 8, LG V50 and Galaxy S8 running Android 9, Pixel 2 XL, LG VELVET and Galaxy s21 running Android 11 as the role of $\mathcal{C}$, Nexus 5x running Android 6 as the role of $\mathcal{A}$, and LG VELVET as the role of $\mathcal{M}$. Since they are running Android, all testing devices in $\mathcal{C}$ provide the 'Enable Bluetooth HCI snoop log' menu which allows recording of the Bluetooth HCI dump log in the background. However, the recorded logs are stored in an inaccessible folder such as 'data/misc/bluedroid/logs.' Thus, instead of directly accessing the location, we leveraged an Android developer option, 'bug report' [22]. Because the bug report allows users to extract the HCI dump logs, we could pull the recorded HCI dump logs from $\mathcal{C}$ devices without any obstacle. From the logs, we could confirm that all of $\mathcal{C}$ left their link keys associated with $\mathcal{M}$ in the HCI dump logs, and could extract them successfully.

After extracting the link keys, we verified the validity of the extracted link keys through the following steps:

1) Change BDADDR of $\mathcal{A}$ to that of $\mathcal{C}$,
2) Install fake bonding information, including the extracted link key for $\mathcal{M}$ on $\mathcal{A}$,
3) Turn off and then turn on Bluetooth on $\mathcal{A}$, and
4) Establish a Bluetooth tethering connection between $\mathcal{A}$ and $\mathcal{M}$, and check whether following LMP authentication succeeds with the fake bonding information—they do not start a new pairing procedure if the key is correct.

To install fake bonding information, we modify the device management file '/data/misc/bluedroid/bt_config.conf' in Nexus 5x. The fake bonding information includes the BDADDR of $\mathcal{M}$, extracted link key, and a list of profile services supported in $\mathcal{M}$. Fig. 10 shows the fake bonding information we added to bt_config.conf, where the link key

```
[48:90:xx:xxx:xx:xx]
Name = VELVET
...
Service = 00001115-0000-1000-8000-00805f9b34fb 00001116-
0000-1000-8000-00805f9b34fb
LinkKey = 71a70981f30d6af9e20adee8aafe3264
```

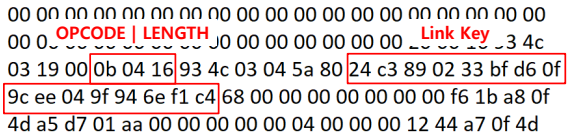
Fig. 10: Fake bonding information for $\mathcal{M}$

'71a70981f3...' is an example of extracted bonded key from the HCI of $\mathcal{C}$. The service is the list of universally unique identifiers (UUID) that are supported in the corresponding device $\mathcal{M}$; 0x00001115 and 0x00001116 are the UUIDs of Bluetooth tethering (PAN profile). Since the roles of both PAN server and client are supported in the Android platform, we leverage the PAN profile to make a profile connection and therefore trigger LMP authentication subsequently in step 5. During the authentication, if the link key is incorrect, the LMP authentication will fail and a new pairing procedure must be initiated; otherwise, the LMP authentication will succeed and the profile connection will be established. Thus, by checking whether the PAN connection is successfully established without any additional pairing procedure, we can validate the correctness of extracted link key.

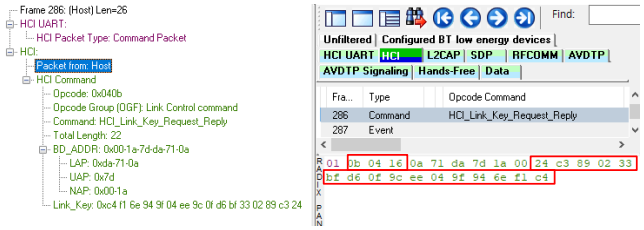
**Link key extraction via USB sniff.** We test two Bluetooth systems as $\mathcal{C}$ running Windows 10 on two PCs. One system consists of CSR harmony host stack and QSENN CSR V4.0 (a USB type Bluetooth controller), and another one consists of Microsoft Bluetooth Driver host stack [26] and QSENN CSR V4.0. Thus, the physical interfaces of HCI of both systems are USB.

The attack procedure is the same as the first experiment, except that we extract HCI data by sniffing the physical interface of HCI (which is USB) using 'Free USB Analyzer' [16] rather than a HCI dump log. When sniffing, the raw data of USB traffic is captured in a binary stream. We thus develop a converter in C that converts the binary stream into a string of hex codes in ASCII format [27]. From the converted data, we found that the USB dump comprises lots of HCI and NULL data. However, as the converted data is in ASCII format, we can simply find the hex data corresponding to the HCI_Link_Key_Request_Reply HCI command. For example, since the command always starts with '0b 04 16' where the first two bytes (0x0b04) indicate the opcode of HCI_Link_Key_Request_Reply and the rest one byte (0x16) is the length value of its payload, we can extract the target link key by searching '0b 04 16' in the converted data as Fig. 11a shows. In the figure, we can see the link key follows six bytes (indicating the address of the peer device) after '0b 04 16', which is '0xc4f16e949f...89c324' in big-endian byte order.

To validate the correctness of the link key extracted from USB sniffing, we compared the link key extracted by USB sniffing on $\mathcal{C}$ and that logged by HCI dump in $\mathcal{M}$. For example, as Fig. 11a and 11b show, we can check the link key from USB sniff on $\mathcal{C}$ and that from HCI dump on $\mathcal{M}$ are the same, thereby we can confirm the link key is correctly extracted.




(a) Link key in USB sniff from $\mathcal{C}$



(b) Corresponding link key from $\mathcal{M}$

Fig. 11: Link keys in HCI data from USB sniff and HCI dump

| Fra... | Type | Opcode Command | Event | Handle | Status |
|---|---|---|---|---|---|
| 323 | Command | HCI_Create_Connection | | | |
| 324 | Event | HCI_Create_Connection | HCI_Command_Status | | Success |
| 325 | Event | | HCI_Connection_Complete | 0x0006 | Success |
| 348 | Command | HCI_Authentication_Requested | | 0x0006 | |
| 350 | Event | HCI_Authentication_Requested | HCI_Command_Status | | Success |
| 352 | Event | | HCI_Link_Key_Request | | |
| 358 | Command | HCI_Link_Key_Request_Negative_Reply | | | |
| 360 | Event | | HCI_IO_Capability_Request | | |

(a) HCI dump for normal pairing

| Fra... | Type | Opcode Command | Event | Handle | Status |
|---|---|---|---|---|---|
| 392 | Event | | HCI_Connection_Request | | |
| 393 | Command | HCI_Accept_Connection_Request | | | |
| 394 | Event | HCI_Accept_Connection_Request | HCI_Command_Status | | Success |
| 395 | Event | | HCI_Connection_Complete | 0x0003 | Success |
| 428 | Command | HCI_Authentication_Requested | | 0x0003 | |
| 429 | Event | HCI_Authentication_Requested | HCI_Command_Status | | Success |
| 430 | Event | | HCI_Link_Key_Request | | |
| 431 | Command | HCI_Link_Key_Request_Negative_Reply | | | |
| 433 | Event | | HCI_IO_Capability_Request | | Success |

(b) HCI dump for pairing under page blocking attack

Fig. 12: HCI dump logs for normal pairing and pairing under page blocking attack

```
[ ./stack/btu/btu_hcif.c]
@@ -165,6 +165,7 @@ void btu_hcif_process_event (...)
            btu_hcif_extended_inquiry_result_evt (p);
            break;
        case HCI_CONNECTION_COMP_EVT:
+           usleep(10000000);
            btu_hcif_connection_comp_evt (p);
            break;
```

Fig. 13: Proof of concept code for PLOC

Additionally, we also examine Linux ubuntu 20.04 to implement the link key extraction attack other than Windows OS. As a result, we find that Bluetooth bonding information is installed in '/var/lib/bluetooth/[MAC Address of device]' folder in Linux. After investigating the folder, we confirm the information file includes the link key of the corresponding device. Moreover, it is also straightforward to implement the link key extraction attack in Linux OS by leveraging both USB sniff and HCI dump log, because there are USB sniffing solutions as well as the bluez-hcidump package. However, running USB sniffing and bluez-hcidump, and accessing the bonding information file in Linux require the superuser privilege. Thus, the practicality of link key extraction in Linux depends on the attacker's affordable privilege.

*2) Page Blocking Attack:* Unlike the link key extraction attack, it is no longer possible to demonstrate the feasibility of page blocking attack only by investigating the user interface information because we cannot confirm whether the subsequent Just Works pairing is caused by our attack, or initiated by $\mathcal{M}$ by accident. Thus, we now analyze the HCI dump log generated in our experiment.

Fig. 12a and Fig. 12b show examples of HCI dump logs captured from $\mathcal{M}$ during an ordinary pairing and a pairing under page blocking attack, respectively. During the normal pairing (Fig. 12a), the host of $\mathcal{M}$ first attempts to establish a connection with $\mathcal{C}$ via HCI_Create_Connection, and executes pairing by HCI_Authentication_Requested, which is the first HCI message for pairing. Then, the controller of $\mathcal{M}$ requests a stored link key by delivering HCI_Link_Key_Request event to the host. However, as $\mathcal{M}$ does not have bonded link key with $\mathcal{C}$ yet, the host replies HCI_Link_Key_Request_Negative_Reply, which implies the absence of the link key. On receipt of it, the controller finally initiates a pairing procedure.

On the other hand, under the page blocking attack, there should be a preceding connection request event on $\mathcal{M}$, HCI_Connection_Request, which is an upstream event from the controller to notify the host that a page request has been received from $\mathcal{A}$. Then, the host accepts the connection request (HCI_Accept_Connection_Request command), thereby a Bluetooth connection between $\mathcal{A}$ and $\mathcal{M}$ is established. After the connection establishment, a pairing procedure is initiated with HCI_Authentication_Requested command in the same manner as the normal pairing. When it is under page blocking attack, $\mathcal{M}$ should be the pairing initiator (HCI_Authentication_Requested command) and the connection responder (HCI_Connection_Request event) simultaneously. Therefore, we can confirm whether our attack is correctly deployed by checking if the HCI dump log is recorded in the same flow as shown in Fig. 12b during the experiment.

For the attack validation, we utilize two Nexus 5x devices running Android 6 as $\mathcal{A}$ and $\mathcal{C}$, and mount the attack against diverse mobile systems $\mathcal{M}$ such as Nexus 5x running Android 8, LG V50 and Galaxy S8 running Android 9, Pixel 2 XL, LG VELVET and Galaxy s21 running Android 11, and iPhone Xs running iOS 14.4.2.

To launch the attack, we first let $\mathcal{A}$ establish a PLOC connection with $\mathcal{M}$. To make both of them stay connected without processing the host layer connection, we make $\mathcal{A}$ postpone the handling of the HCI event sent from the controller, thus $\mathcal{A}$ does not process the next steps for the host layer connection. Update of Bluetooth connection state begins when the host receives an HCI_Connection_Complete event from the controller. In Nexus 5x, 'btu_hcif_process_event()', a callback function for HCI events in bluedroid, processes HCI_Connection_Complete events. Thus, we implement the PLOC condition in $\mathcal{A}$ by making the host postpone the

| OS | Host stack | Device | SU privilege |
|---|---|---|---|
| Android 8 | Bluedroid | Nexus 5x | N |
| Android 9 | Bluedroid | LG V50 | N |
| Android 9 | Bluedroid | Galaxy S8 | N |
| Android 11 | Bluedroid | Pixel 2 XL | N |
| Android 11 | Bluedroid | LG VELVET | N |
| Android 11 | Bluedroid | Galaxy s21 | N |
| Windows 10 | Microsoft Bluetooth Driver | QSENN CSR V4.0 | N |
| Windows 10 | CSR harmony | QSENN CSR V4.0 | N |
| Ubuntu 20.04 | BlueZ | QSENN CSR V4.0 | Y |

TABLE I: List of tested devices that are vulnerable to link key extraction attack

handling of the event in btu_hcif_process_event().

In a page blocking attack, the PLOC state shall be held until a pairing procedure is initiated by $\mathcal{M}$ theoretically. In practice, it may require a bit complex code implementation in the library because of timing synchronization as well as exception handling issues. For example, the host should stop the postponement when a pairing procedure is initiated by $\mathcal{M}$ and handle other HCI commands and events that occur during its PLOC state. It also requires the prevention of link drop, which may be caused by connection timed-out. However, it can be solved by exchanging some dummy data, such as SDP query. Our experiment assumed that $\mathcal{M}$ initiates a pairing procedure in 10 seconds after establishing the PLOC connection for simplicity. Thus, we do not have to care the aforementioned exceptions and simplify the implementation code in the experiment. Fig. 13 shows our proof of concept (PoC) implementation that makes the host stay in a PLOC state for a fixed duration (10 seconds) before calling btu_hcif_connection_comp_evt(). After the duration, $\mathcal{C}$ will exit from the PLOC state; and if $\mathcal{M}$ initiates a pairing as we assumed, it carries out a pairing procedure.

With the PoC code, we evaluate our page blocking attack following the attack procedure in Section V against the mobile devices $\mathcal{M}$, and after the pairing, we check whether the HCI dump log recorded during the test is the same as Fig. 12b.

### C. Evaluation Results

*1) Results for Link Key Extraction Attack:* The experiment of HCI dump on Android devices with versions 8, 9, and 11 confirms that the extracted link keys are the same as the existing keys shared between $\mathcal{C}$ and $\mathcal{M}$, and the LMP authentication succeeded, demonstrating they are all vulnerable to our link key extraction attack. Additionally, according to the experiment of USB sniff, we also confirm that the tested PC systems (one with CSR harmony and QSENN CSR V4.0, and the other with Microsoft Bluetooth Driver host stack and QSENN CSR V4.0), are all vulnerable to link key extraction attack. The complete list of tested devices vulnerable to the link extraction attack is given in Table I, where the rightmost column means whether the attacker requires a superuser privilege.

| Device | Success rate without page blocking | Success rate with page blocking |
|---|---|---|
| iPhone Xs (iOS 14.4.2) | 52% | 100% |
| Nexus 5x (Android 8) | 52% | 100% |
| LG V50 (Android 9) | 57% | 100% |
| Galaxy S8 (Android 9) | 42% | 100% |
| Pixel 2 XL (Android 11) | 60% | 100% |
| LG VELVET (Android 11) | 60% | 100% |
| Galaxy s21 (Android 11) | 51% | 100% |

TABLE II: Success rates of MITM connection establishment

*2) Results for Page Blocking Attack:* According to our experiment with each mobile device, we observed that all of the HCI dump logs captured in $\mathcal{M}$ are the same as Fig. 12b (for iPhone, we analyzed dump log from $\mathcal{A}$ instead of $\mathcal{M}$ since it does not provide HCI dump), which implies the pairing is conducted between $\mathcal{M}$ and $\mathcal{A}$ (rather than $\mathcal{C}$). Therefore, we confirm that our attack succeeded, and all of the mobile devices we tested are vulnerable to our page blocking attack. The detailed experiment results showing the comparative success rates of establishing MITM connections with/without page blocking attack are given in Table II. For the normal case without page blocking, we establish connections for each device 100 times, and calculate the success rate. As a result, we observed 42$\sim$60% of success rate, implying the establishment of MITM connections with the target devices is quite random. Whereas, under our page blocking attack where $\mathcal{A}$ becomes a connection initiator, we observed 100% of success rate, implying page blocking attack allows the establishment of MITM connections entirely as the attacker's intention.

## VII. Mitigations

### A. Link Key Extraction Attack

The root cause of vulnerability to the link key extraction attack is that the link key is transferred via HCI as plaintext, and HCI data can easily be leaked via HCI dump or hardware interface (e.g., USB) that typical users can easily access.

The first mitigation is to filter out link keys from the HCI dump log. The solution can be implemented by enabling the HCI dump module to monitor HCI headers; if the monitored HCI message includes a message related to link keys, the dump module logs only HCI header, not its payload. For example, in Fig. 3, the HCI packet of HCI_Link_Key_Request_Reply command in RADIX is '01 0b 04 16 96 55 46 6d ...', where the first byte (0x01) indicates that the packet is an HCI command, and the next three bytes (0x0b0416) are the HCI header composed of operation code (0x0b04) and payload length (0x016 = 22 bytes) as described in Section VI-B1. Thus, when HCI dump meets an HCI packet that starts with 0x010b0416, it may omit to record the payload into its dump log by logging only the first four bytes of the header or replace the link key with a random value.

Another mitigation is to encrypt the payload of HCI packets related to link keys. An attacker may extract HCI data from

UART physical line, or by sniffing USB data, which can break the first mitigation. If the HCI payload is encrypted, link keys can be secured even if such physical attacks extract the HCI data. However, encrypting the payload of HCI packets may require major updates or revision of current specifications to include such encryption and key exchange functionalities between the host and the controller.

### B. Page Blocking Attack

According to GAP, $\mathcal{A}$ should initiate the pairing and LMP authentication if any subsequent connecting services request security mechanisms such as authentication, authorization, and encryption; otherwise, they can be skipped. For example, $\mathcal{A}$ is allowed to connect to SDP service in $\mathcal{M}$ without LMP authentication, because SDP service does not require the enforcement of security mechanisms. For this reason, it is difficult to predict whether any pairing will occur during a session in practice. Thus, the specification allows either a connection initiator ($\mathcal{A}$) or a responder ($\mathcal{M}$) to be a pairing initiator in the middle of the session. By leveraging this, the page blocking attack allows $\mathcal{A}$ to become a connection initiator, and $\mathcal{M}$ to become a pairing initiator. From the user's point of view, it is indistinguishable between normal pairing and pairing under a page blocking attack since they are processed the same on the user interface. The other case where $\mathcal{A}$ and $\mathcal{M}$ play the opposite roles can normally occur as well, thus simply checking each role of the pairing initiator and the connection initiator cannot correctly detect our page blocking attack. Therefore, one effective mitigation strategy is to check the roles of the pairing initiator and connection initiator and to check whether the IO capability of the connection initiator is NoInputNoOutput. If it is the case, we need to make sure the victim drops the pairing or re-initiates the pairing again in another safer mode.

## VIII. Related Work

Antonioli et al. [8] introduced an attack on key agreement protocol of Bluetooth, called KNOB attack. It makes victims agree on a key with one byte of entropy. In order to implement the attack, the authors reversed a firmware of BCM4339 Bluetooth chipset and installed their customized attack operations in the controller. Antonioli et al. [7] also introduced another attack called BIAS that exploits one-way authentication vulnerability of Secure Simple Pairing, and perform downgrade attack on secure authentication procedure for Secure Connection. BIAS attack also requires a modification of controller for the purpose of installing customized LMP operations. For general users, such reversing and modification procedures of controllers would be impediments to the achievement of the attacks in practice. Contrarily, our attacks just require to implement attack operations above the controller layer, therefore, our attacks are more practical attacks.

Seri et al. [5] introduced security vulnerability of Bluetooth implementations, called BlueBorne, in diverse operating systems such as Android, iOS, and Windows. Its attack vectors allow an attacker to silently create unauthenticated BR/EDR connections and install malicious codes on the victim devices. One precondition required for mounting BlueBorne attack is to only activate Bluetooth in the victim device. Our link key extraction attack also silently works against a victim device, but manual access to the corresponding paired device is required.

For MITM attacks, Sun et al. [6] introduced a vulnerability that allows an MITM attack against the passkey entry SSP association mode. Sharmila et al. [24], Hypponen et al. [2], Haataja et al. [1] presented MITM attacks that downgrade SSP to Just Works association mode. Melamed et al. [28] introduced another MITM attack on Bluetooth connections between a Mobile App and a Bluetooth Peripheral device. Zhang et al. [29] demonstrated that MITM attacks are possible due to the lack of detailed programming guideline for a Secure Connections Only (SCO) mode in the specification. In our study, we focused on how to ensure the establishment of MITM connections by the attacker, and subsequently execute SSP downgrade attack, which the previous MITM attacks simply assumed as a precondition for their attacks.

## IX. Conclusion

In this paper, we present two novel attacks on the authentication of Bluetooth BR/EDR, which are link key extraction and page blocking attacks. The link extraction attack exploits a vulnerability where link keys are logged into an HCI dump file in plaintext and can be easily extracted in practice. Once a link key is extracted, the attacker can continuously leverage it to eavesdrop on Bluetooth communications protected by the link key and establish other impersonated connections, breaking the LMP authentication of bonded devices. Page blocking attack aims to control the victim's connection such that it is forced to be established with the attacker's device as his intention in a deterministic manner, which is practically difficult to control due to the unpredictability of arbitrary session establishments in practice. The page blocking attack exploits the laxity of the specification - the connection initiator does not have to be a pairing initiator. That is, if a Bluetooth connection is established by the attacker and the victim then triggers a pairing procedure, the attacker can establish a malicious connection and make it paired in Just Works association model initiated by the victim, breaking SSP authentication of non-bonded devices. Since our attacks are standard-compliant and can be delivered above the controller layer without firmware modification, they can be deployed without much difficulty in practice, demonstrating our attacks can pose real threats to Bluetooth security.

## Acknowledgment

This work was supported by IITP grant funded by the MSIT, Korea (No.2019-0-00533, IITP-2022-2020-0-01819, IITP-2021-0-01810) and Basic Science Research Program through the National Research Foundation funded by the Ministry of Education, Korea(NRF-2021R1A6A1A13044830).

## REFERENCES


[1] K. Haataja and P. Toivanen, "Two Practical Man-in-the-Middle Attacks on Bluetooth Secure Simple Pairing and Countermeasures," *IEEE Transactions on Wireless communications*, vol. 9, no. 1, pp. 384–392, 2010.

[2] K. Hypponen and K. M. Haataja, ""Nino" Man-in-the-Middle Attack on Bluetooth Secure Simple Pairing," in *Proceedings of the IEEE/IFIP International Conference in Central Asia on Internet*, 2007, pp. 1–5.

[3] T. R. Mutchukota, S. K. Panigrahy, and S. K. Jena, "Man-in-the-Middle Attack and its Countermeasure in Bluetooth Secure Simple Pairing," in *Proceedings of the International Conference on Information Processing*, 2011, pp. 367–376.

[4] K. Saravanan, L. Vijayanand, and R. Negesh, "A Novel Bluetooth Man-in-the-Middle Attack based on SSP using OOB Association Model," *arXiv preprint arXiv:1203.4649*, 2012.

[5] B. Seri and G. Vishnepolsky, "BlueBorne™, The Dangers of Bluetooth Implementations: Unveiling Zero Day Vulnerabilities and Security Flaws in Modern Bluetooth Stacks," ARMIS, 2017, https://www.armis.com/blueborne/, Accessed: 2021-10-12.

[6] D.-Z. Sun, Y. Mu, and W. Susilo, "Man-in-the-Middle Attacks on Secure Simple Pairing in Bluetooth Standard V5. 0 and its Countermeasure," *Personal and Ubiquitous Computing*, pp. 55–67, 2018.

[7] D. Antonioli, N. O. Tippenhauer, and K. Rasmussen, "BIAS: Bluetooth Impersonation Attacks," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2020, pp. 549–562.

[8] D. Antonioli, N. O. Tippenhauer, and K. B. 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 SEC)*, 2019, pp. 1047–1061.

[9] G. Wassermann, "Bluetooth Implementations may not Sufficiently Validate Elliptic Curve Parameters during Diffie-Hellman key Exchange," https://www.kb.cert.org/vuls/id/304725/, Accessed: 2021-10-12.

[10] E. Biham and L. Neumann, "Breaking the Bluetooth Pairing-Fixed Coordinate Invalid Curve Attack," 2018, http://www.cs.technion.ac.il/~biham/BT/bt-fixed-coordinate-invalid-curve-attack.pdf, Accessed: 2021-10-12.

[11] BlueZ, "Linux Open Source for the Bluetooth Host Stack," http://www.bluez.org/about/, Accessed: 2021-10-12.

[12] AOSP, "Bluetooth Host Stack in Android Open Source Project," https://source.android.com/devices/bluetooth, Accessed: 2021-10-12.

[13] B. SIG, "Bluetooth Core Specification v5.3," https://www.bluetooth.com/specifications/specs/core-specification/, Accessed: 2021-10-12.

[14] D. Hulton, "btpincrack," 2006, http://openciphers.sourceforge.net/oc/btpincrack.php, Accessed: 2021-10-12.

[15] Y. Shaked and A. Wool, "Cracking the Bluetooth Pin," in *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2005, pp. 39–50.

[16] H. Software, "Free USB Analyzer," https://freeusbanalyzer.com/, Accessed: 2021-10-12.

[17] M. Cominelli, F. Gringoli, P. Patras, M. Lind, and G. Noubir, "Even black cats cannot stay hidden in the dark: Full-band de-anonymization of bluetooth classic devices," pp. 534–548, 2020, 2020 IEEE Symposium on Security and Privacy (SP).

[18] AOSP, "Android Bluetooth Verifying and Debugging," https://source.android.com/devices/bluetooth/verifying_debugging, Accessed: 2021-10-12.

[19] BlueZ, "HCI Data Dumper in the BlueZ Project," http://www.bluez.org/?s=hcidump, Accessed: 2021-10-12.

[20] N. W. Group, "Snoop Version 2 Packet Capture File Format," https://datatracker.ietf.org/doc/html/rfc1761, Accessed: 2021-10-12.

[21] A. developers, "Configure On-Device Developer Options," https://developer.android.com/studio/debug/dev-options, Accessed: 2021-10-12.

[22] ——, "Capture and Read Android Bug Reports," https://developer.android.com/studio/debug/bug-report, Accessed: 2021-10-12.

[23] Frontline, "ComProbe USB 2.0 Protocol Analyzer - FTS4USB™," https://fte.com/products/FTS4USB-HCI.aspx, Accessed: 2021-10-12.

[24] D. Sharmila, R. Neelaveni, and K. Kiruba, "Notice of Violation of IEEE Publication Principles: Bluetooth Man-In-The-Middle Attack based on Secure Simple Pairing using Out Of Band Association Model," in *Proceedings of the IEEE International Conference on Control, Automation, Communication and Energy Conservation*, 2009, pp. 1–6.

[25] AOSP, "Android Codenames, Tags, and Build Numbers," https://source.android.com/setup/start/build-numbers, Accessed: 2021-10-12.

[26] Microsoft, "Bluetooth Driver Stack," https://docs.microsoft.com/en-us/windows-hardware/drivers/bluetooth/bluetooth-driver-stack, Accessed: 2021-10-12.

[27] "BinaryToHex," https://github.com/changsuck1/BinaryToHex, 2021.

[28] T. Melamed, "An Active Man-in-the-Middle Attack on Bluetooth Smart Sevices," *Safety and Security Studies*, vol. 15, p. 2018, 2018.

[29] Y. Zhang, J. Weng, R. Dey, Y. Jin, Z. Lin, and X. Fu, "Breaking Secure Pairing of Bluetooth Low Energy using Downgrade Attacks," in *Proceedings of the USENIX Security Symposium (USENIX SEC)*, 2020, pp. 37–54.