ERNW
providing security.

ERNW WHITE PAPER 74

AIROHA RACE: BLUETOOTH HEADPHONE VULNERABILITIES

Version:	1.0
Date:	December 27, 2025
Classification:	Public

ERNW
providing security.

Table of Content

1 Handling

The present document is classified as *Public*. Any distribution or disclosure of this document REQUIRES the permission of the document owner as referred in Section *Document Status and Owner*.

1.1 Document Status and Owner

As the owner of this report, the document owner has exclusive authority to decide on the dissemination of this document and responsibility for the distribution of the applicable version in each case to the places.

The possible entries for the status of the document are *Initial Draft*, *Draft*, *Effective* and *Obsolete*.

Report Information	
Title:	ERNW White Paper 74 - Airoha RACE: Bluetooth Headphone Vulnerabilities
Document Owner:	ERNW Enno Rey Netzwerke GmbH
Version:	1.0
Status:	Effective
Classification:	Public
Project Number:	-
Author(s):	Dennis Heinze, Frieder Steinmetz

Table 2: Document Status and Owner

1.2 Classification Levels

Classification Level	Audience
Public:	Everyone
Internal:	All employees and business partners
Confidential:	Only employees
Secret:	Only selected employees

Table 3: Classification Levels

1.3 Document Version History

Version	Date	Details
1.0	December 27, 2025	Initial version after quality assurance.

Table 4: Document Version History

2 Introduction

Bluetooth headphones and earbuds are amongst the most widely used Bluetooth peripherals. As such, they are an interesting target for security research. What is possible when such a device is compromised? What could an attacker achieve? Are Bluetooth headphones a target that's worth putting in time and resources? After initially discovery of an interesting target we were asking ourselves these questions.

During our research we identified three vulnerabilities in a common set of Bluetooth SoCs (Systems on a Chip) that are used in a variety of Bluetooth headphones and earbuds. In this white paper, we provide a write-up of the vulnerabilities and discuss the difficulties of disclosure and patching. At the end, we try to derive general insights for the questions above.

Airoha [1] is a vendor that, amongst other things, builds Bluetooth SoCs and offers reference designs and implementations incorporating these chips. They have become a large supplier in the Bluetooth audio space, especially in the area of True Wireless Stereo (TWS) earbuds. Several reputable headphone and earbud vendors have built products based on Airoha's SoCs and reference implementations using Airoha's Software Development Kit (SDK).

In previous Bluetooth related research we stumbled upon a pair of these headphones. During the process of obtaining the firmware for further research we initially discovered the custom Bluetooth protocol called *RACE*. We found that this protocol has various powerful capabilities, such as allowing us to read from and write to arbitrary locations in both flash and RAM. Additionally, it was exposed over Bluetooth BR/EDR (also known as Bluetooth Classic) and Bluetooth Low Energy. In both cases, pairing is not required. As such, an attacker within Bluetooth range can connect to the audio device and manipulate RAM and flash contents without prior authentication.

The lack of pairing does not only enable unauthenticated attackers to use the RACE protocol. It constitutes a vulnerability on its own. An attacker can connect to the vulnerable audio device and, for example, use the Bluetooth Hands-Free Profile (HfP) to send or receive audio. This might allow eavesdropping via the device's microphone.

Along with this white paper, we release *race-toolkit*, a tool for researchers to further work on Airoha based devices. We also release instructions for users to check whether their device might be affected or still vulnerable (see Section 6).

We also want to thank Airoha for their great cooperation during their first disclosure process. Their advisory can be found in the Airoha Product Security Bulletin [2].

3 Background

This section will introduce the required background for the following vulnerability write-up and discussion. If you're already familiar with Bluetooth you might skip this section.

3.1 Headphones and Earbuds

This white paper mentions different types of audio devices. Generally, the Airoha SoCs can be used in various types of audio devices, both for transmitting and playing audio. Throughout this white paper the terms headphones and earbuds are used. However, these issues can also apply to Bluetooth speakers or transmitters.

One important term here is True Wireless Stereo (TWS). It describes earbuds that are truly wireless, i.e., the left and right earbud or speaker are not connected. Compared to traditional over ear headphones, these devices are more complex. Both earbuds (left and right) need to have a Bluetooth connection. Moreover, they also need to stay in sync. This is one of the markets where Airoha SoCs are prevalent.

3.2 Bluetooth Classic vs Bluetooth Low Energy

Bluetooth defines two distinct protocol families: Bluetooth BR/EDR (mostly referred to as Bluetooth Classic) and Bluetooth Low Energy (BLE). Both operate on the same 2.4 GHz ISM band, but have two distinct physical layers and differ in the layers above. On the application layer they share some protocols (such as L2CAP).

Bluetooth Classic traditionally provided connection-oriented communication designed for higher throughput, such as audio. Although this is starting to change with the introduction of LE Audio.

BLE was introduced with the Bluetooth Core specification 4.0. It is optimized, as the name implies, for low energy consumption. It added several new concepts, such as Advertisements or the Generic Attribute Profile (GATT), which provides a structured key-value interface over the Attribute Protocol (ATT).

Although Bluetooth Classic and BLE can coexist in a single controller, they are logically independent stacks and use separate security procedures, addressing schemes, and data transport mechanisms.

3.3 Bluetooth Stack

The Bluetooth stack follows a host–controller separation, defined to isolate lower-level radio and link management from higher-level protocol and application logic. This split allows Bluetooth host implementations to operate with different Bluetooth chipsets and controllers via a standardized interface, the Host Controller Interface (HCI).

At the lower end, the Controller implements the Physical Layer (PHY) and Link Layer (LL/LCP) for BLE, or the Baseband and Link Manager (LM/LMP) for Bluetooth Classic. The controller typically resides on the Bluetooth hardware, often as a discrete SoC or module.

The Host comprises higher layers such as the L2CAP, the Attribute Protocol (ATT) and Generic Attribute Profile (GATT) in BLE, and protocols like RFCOMM, SDP, and the various application profiles in Bluetooth Classic.

Communication between the two components occurs over the previously mentioned HCI. HCI can be implemented via various transport protocols, such as USB, or UART.

The graphic below shows a simplified high-level overview of the Bluetooth stack. It only includes protocols and parts that are important for this white paper. The blue components are BLE components, the green ones related to Bluetooth Classic.

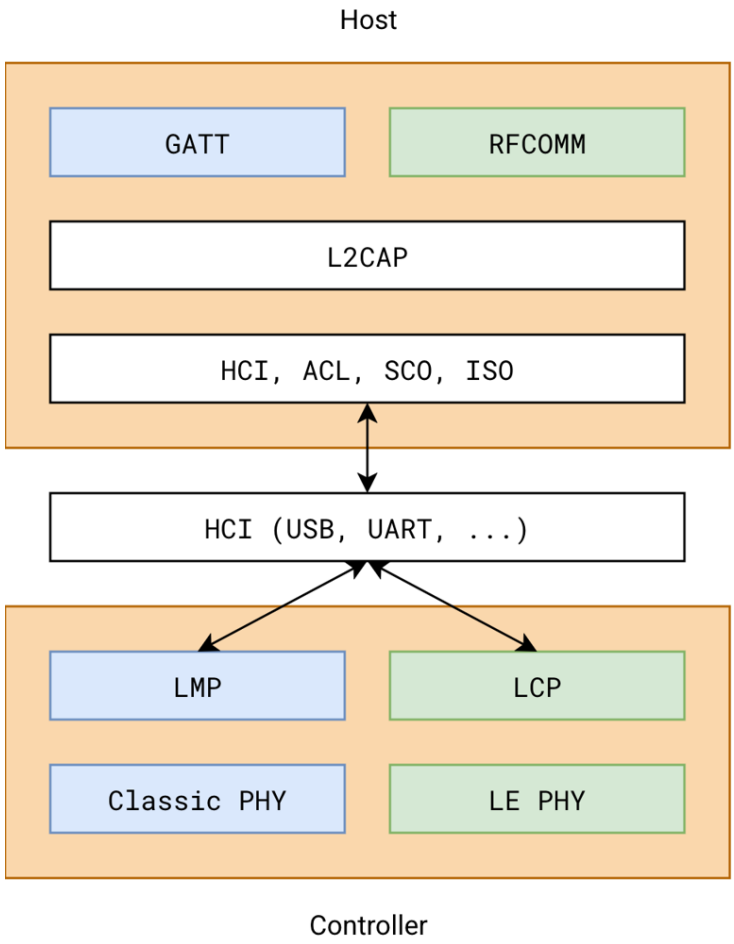

Figure 1: High-Level Bluetooth Stack

3.4 Pairing and Bonding

Bluetooth devices use *pairing* to establish shared key material for authentication and encryption. In Bluetooth Classic, pairing occurs during connection establishment through the Link Manager Protocol (LMP). The output of pairing is a *Link Key*. This Link Key is stored on the Host and provided to the controller whenever a secure connection is to be established.

In BLE, pairing is handled by the Security Manager Protocol (SMP) operating over a fixed L2CAP channel. The current state of the art is using LE Secure Connections, which perform an Elliptic Curve Diffie–Hellman key exchange to derive the Long-Term Key (LTK) used for link-layer encryption.

One major difference here is that in BLE, the pairing is performed on the host part of the stack, in Bluetooth Classic on the controller part.

Bonding refers to the storage of these keys so that the devices can reconnect without repeating the pairing procedure. Bluetooth Classic and BLE maintain their own separate keys and addressing information.

3.5 Bluetooth Addressing

Every Bluetooth device is identified at the Link Layer by a Bluetooth Device Address (often referred to as *BD_ADDR*). This address works similarly to MAC addresses in other wireless systems. The Bluetooth Device Address is six bytes long for both Bluetooth Classic and BLE. However, there are some differences in addressing in these two technologies.

In Bluetooth Classic, the address is (or should be) globally unique and assigned by the manufacturer. The first 24 bits of the address contain a manufacturer identifier (the OUI - Organizationally Unique Identifier), the other bits are device specific. This address is static and is required to establish a connection with the device. Knowledge of the device address can sometimes be beneficial for an attacker. Even when a device is not discoverable, the knowledge of the address is enough to establish a connection if the device is connectable. The static nature of the address can also become an issue in regard to privacy.

BLE supports various types of addresses. A Public Device Address that is similar to the BD_ADDR in Bluetooth Classic or a type of Random Device Addresses.

A Random Device Address may be either static or private. Private addresses are regenerated periodically to prevent long-term tracking and come in two forms:

- Resolvable Private Addresses (RPA), which include a hash derived from the device's Identity Resolving Key (IRK). RPAs can be recognized by paired devices but appear random to others.
- Non-Resolvable Private Addresses (NRPA), which are generated randomly and cannot be resolved by any party, used primarily for anonymous advertising.

In dual-mode devices that support both Bluetooth Classic and BLE, these addressing systems are independent. A single physical device therefore typically has two addresses: one Classic BD_ADDR and one BLE address (public or random).

3.6 RFCOMM

RFCOMM provides a byte-stream transport that emulates a serial interface over Bluetooth Classic. It operates on top of L2CAP. RFCOMM supports multiple logical data channels multiplexed over a single Bluetooth connection. Channels can be identified via their UUID. RFCOMM often serves as a transparent transport for applications expecting a serial-like interface over a Bluetooth connection.

3.7 GATT

In BLE, application data is mainly exchanged through the Generic Attribute Profile (GATT). GATT operates on top of the Attribute Protocol (ATT), which provides basic operations, such as *Read*, *Write*, or *Notify* over a fixed L2CAP channel.

A GATT server can expose multiple services that are identified by a UUID. Each service in turn can hold one or more characteristics that are also identified by a UUID. The characteristics can have different access configuration, such as whether they are readable, writable, subscribable, or whether a secure connection is required to interact with them.

A GATT client can discover and enumerate available services and interact with them by issuing ATT requests or subscribing to notifications.

In real-world applications, GATT often serves the purpose of an underlying layer for a byte-stream transport (similar to RFCOMM or L2CAP in Bluetooth Classic). This stems from the fact that, in BLE, RFCOMM or L2CAP are not always directly accessible on the application layer. To establish a byte-stream transport, a characteristic that allows reading and writing, or subscribing and writing is created. Sending data then works by writing to the characteristic, receiving data by receiving subscription notification.

3.8 Bluetooth Profiles

Bluetooth profiles define standardized behavior for specific Bluetooth-related applications. A profile specifies how protocols are used, configured and how the devices are expected to behave in a specific scenario. Essentially, they ensure interoperability between devices from different manufacturers.

One example of a Bluetooth profile is the Hands-Free Profile (HFP). It is a Bluetooth Classic profile used for Audio communication. It defines an Audio Gateway (AG) (e.g. a mobile phone) and a Hands-Free unit (HF) (e.g. a headset or a head-unit system inside a car).

HFP builds on RFCOMM to transport control commands using an AT command syntax. These commands are used to manage call setup and termination, audio routing, volume control, and carrier information (such as obtaining the phone number of the device).

4 The RACE Protocol

In this section, the proprietary RACE protocol and the interfaces that the protocol can be communicated over will be described. As all the information about the protocol were obtained by reverse-engineering either firmware, or various companion software, there might be technical inaccuracies or false naming.

A reimplementation of the described transports and protocol messages can be found in the `race-toolkit` GitHub repository [8]. Further technical and practical details and documentation can be found in the tool's documentation.

4.1 Discovery

Prior to this research, we experimented with the new Bluetooth Auracast feature [7], introduced with the Bluetooth Core specification 5.2. For this, we bought an Auracast transmitter device [14], and a pair of Bluetooth headphones that had Auracast capabilities.

We first encountered the RACE protocol while examining USB traffic between the Bluetooth Auracast transmitter device and the corresponding firmware update and configuration utility. This tool allows configuring the device. For example, the Auracast stream name and password can be set. It can also retrieve version information and update the device's firmware.

The MoerLab Auracast Transmitter PC configuration tool can be seen in the following screenshot:

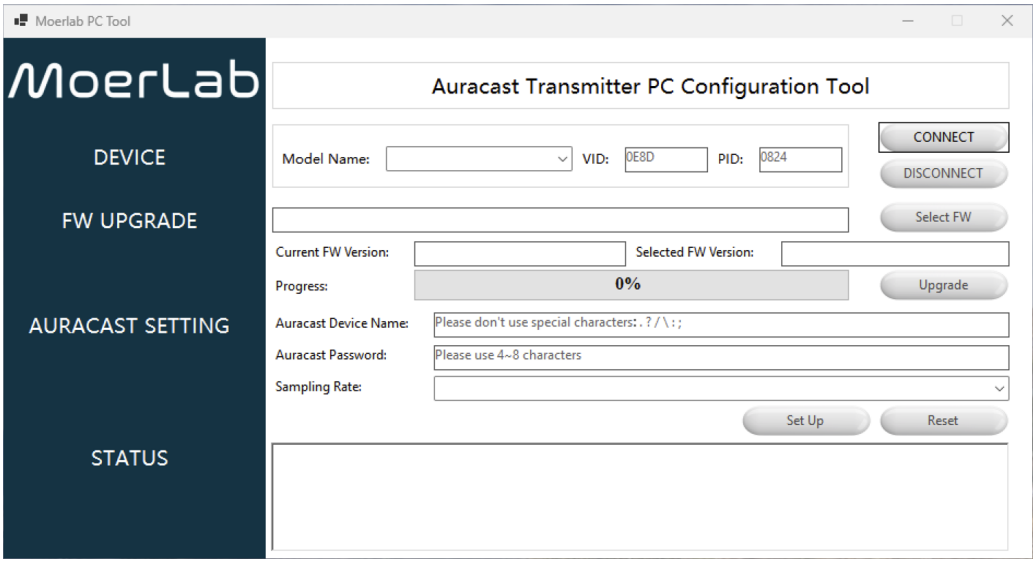

Figure 2: Moerlab Transmitter Tool

The screenshot below shows an excerpt of a Wireshark trace where we identified the USB HID based RACE communication. In this case, the response to the *get buildversion* command is shown.

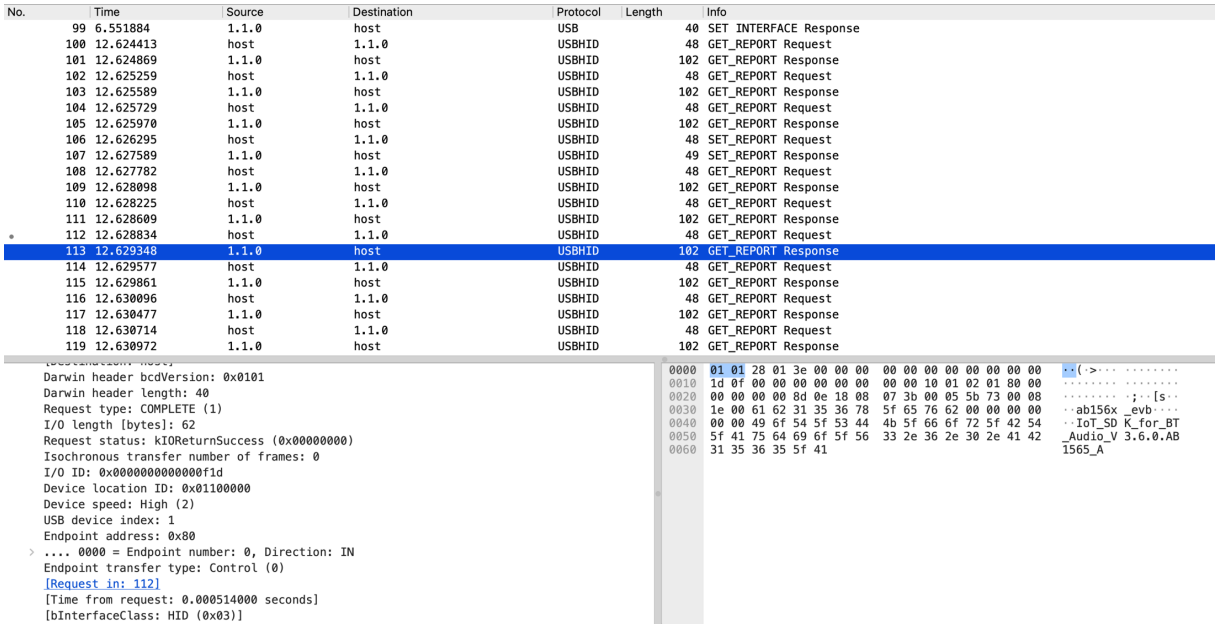

Figure 3: Wireshark HID

An examination of the USB traffic exposed a structure in the communication. The recurring bytes and the structure led to the assumption that this is likely a custom protocol that the application and the device communicate with. The beginning of the packets were mostly the same (`15 5A 0e 02 02 04`), then there was an incremental value (e.g., `00 30 26 00` or `0x263000` in little endian) followed by a larger binary blob.

A correlation of the binary blob in each of the packets reveals that these are fragments of the firmware update file. The incremental part of the packets might either be a counter or the target address.

This can be observed in the following graphic:

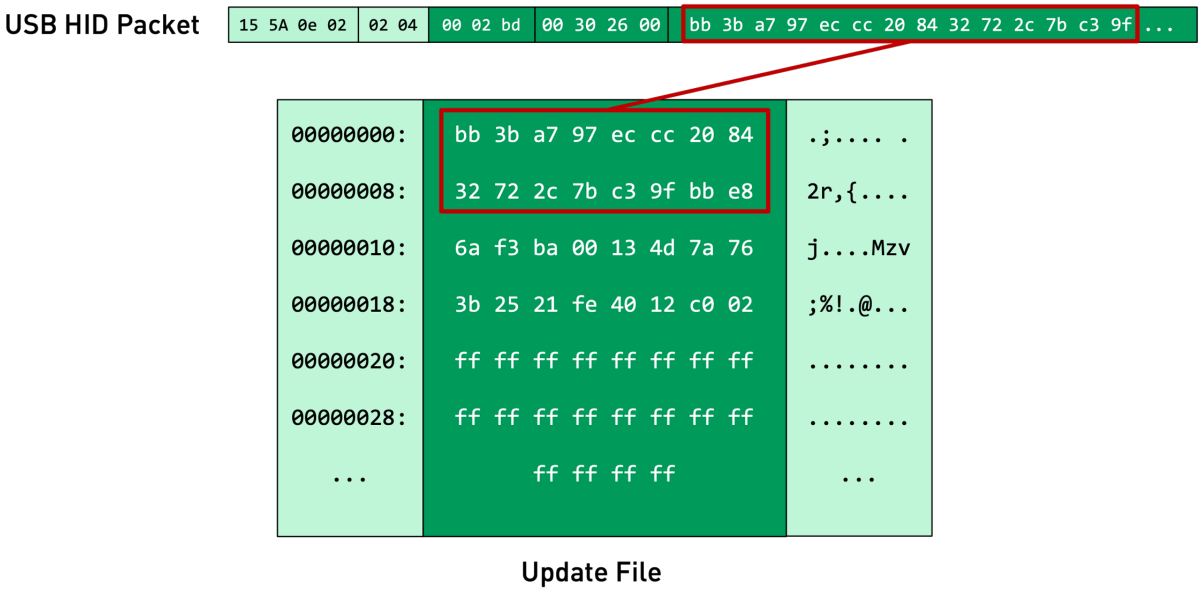

Figure 4: HID FOTA Bytes

An analysis of the firmware update file, the configuration utility, as well as multiple headphone companion apps, helped to gain a better understanding of the protocol. Some of the command IDs had corresponding symbols (in case of the mobile apps) or log strings, that indicated what their purposes might be.

Airoha firmware update files, also known as FOTA, are compressed and sometimes encrypted. The airoha-firmware-parser [15] project implements the decompression which is required to start reverse-engineering the firmware.

4.2 General Protocol Details

The general packet structure of the RACE protocol is shown below.

Head	Type	Length	CMD	Data
1 byte	1 byte	2 bytes	2 bytes	*Length* bytes

Figure 5: RACE Packet Structure

RACE packets start with a 6-byte header. The first byte seems to be mostly the static. It might be a version indicator or just a general RACE protocol packet indicator. In practice, we only observed the values `0x05` and `0x15`, while `0x05` was used during normal communication and `0x15` for firmware update related communication.

The type field can hold different values. It indicates whether the packet is a request or a response. It can also indicate whether the request expects a response. Most packets we observed do expect a response and set type to `0x5a`. Responses have the type `0x5b`.

The last field of the header is the command ID which determines the contents and structure of the rest of the packet. The figure below shows a sample packet with command ID `0x403` (flash read).

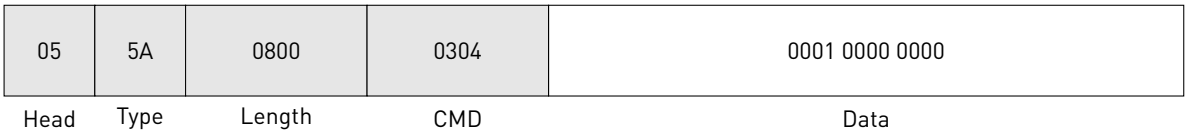

Figure 6: RACE Sample: Read Flash

4.3 Race Commands

Depending on the command ID, the *data* field can have different layouts. Some of the relevant commands are described in this section.

The table below shows a list of commands that we obtained from either reverse-engineering the firmware, or analyzing companion apps and companion software. In the following, some of the commands are outlined in some more detail.

Command ID	Name	Description
`0x1e08`	`GET_BUILD_VERSION`	Get build version.
`0x0403`	`STORAGE_PAGE_READ`	Read Flash.
`0x1680`	`READ_ADDRESS`	Read RAM.
`0x1681`	`WRITE_ADDRESS`	Write RAM.
`0x0cd5`	`GET_BD_ADDRESS`	Get Bluetooth Device Address.

4.3.1 `GET_BUILD_VERSION`

The `GET_BUILD_VERSION` command is very useful for fingerprinting device models and firmware versions. It reveals the SDK version, the SoC model, sometimes the vendor name, and the build date of the firmware. The command does not take any arguments, so the data part of the RACE message is empty.

The command returns a serialized message that contains the various strings that are part of the build version. The layout is as follows:

SoC Name (16 bytes)
SDK Version (48 bytes)
Build Date (48 bytes)

Figure 7: GET_BUILD_VERSION Response

There are three fixed-size compartments. One for the SoC model name that's inside the device, one for the SDK version, and one for a time stamp which is likely the build date.

The screenshot below shows an example of a `GET_BUILD_VERSION` command's response, with the three compartments indicated.

```
❯ hexdump -C wh720n_buildversion.bin
00000000  6d 74 32 38 32 32 78 5f  65 76 6b 00 00 00 00 00  |mt2822x_evk.....|
00000010  4d 54 32 38 32 32 5f 53  44 4b 5f 53 6f 6e 79 2d  |MT2822_SDK_Sony-|
00000020  45 52 36 39 5f 6d 64 72  31 34 5f 63 34 32 73 70  |ER69_mdr14_c42sp|
00000030  5f 31 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |_1..............|
00000040  32 30 32 34 2f 30 39 2f  31 38 20 31 38 3a 35 38  |2024/09/18 18:58|
00000050  3a 35 35 20 47 4d 54 20  2b 30 38 3a 30 30 00 00  |:55 GMT +08:00..|
00000060  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000070
```

Figure 8: Buildversion Hexdump

Concatenating the compartments and removing the null bytes results in a single string that can be used to identify devices. Below are a few examples of concatenated Build Version responses:

- **Marshall ACTON III**:
  - `ab156x_evbV7.0.92023/08/07 21:51:11 GMT +08:00`
- **Sony WH-CH720N**:
  - `mt2822x_evkMT2822_SDK_Sony-ER69_mdr14_c42sp_12024/06/28 13:44:31 GMT +08:00`
- **Jabra Elite 8 Active**:
  - `ab158x_evbIoT_SDK_for_BT_Audio_V3.3.0.AB1585_AB15882024/03/13 20:44:32 GMT +08:00`

4.3.2 STORAGE_PAGE_READ

The `STORAGE_PAGE_READ` command takes three arguments: a storage type, a size, and an address. We found that for many devices the size parameter is ignored and one full flash page (`0x100` bytes) is returned. Some other devices require the size argument. It is important to note that the size is not given in bytes, but apparently in pages. For example, `0x100` bytes are encoded as `0x01`. The number of bytes is shifted 8 bits to the right (e.g. `0x100 >> 8 = 0x01`).

The layout of the command is as follows:

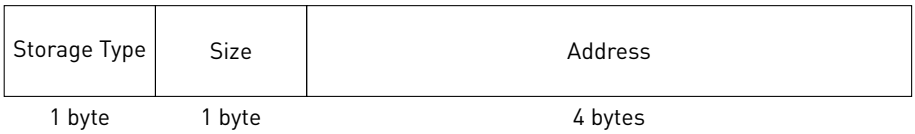

Figure 9: RACE STORAGE_PAGE_READ Command

This command is useful for dumping the entire flash content of the device, which allows for an extraction of the entire firmware and other configuration data such as firmware encryption keys or the Bluetooth device connection table (more on that in Section 5.3).

4.3.3 READ_ADDRESS & WRITE_ADDRESS

The `READ_ADDRESS` and `WRITE_ADDRESS` commands are extremely powerful. Essentially, they allow reading and writing to arbitrary memory addresses on the system. This enables read and write access to RAM, but also to the SoCs registers and peripherals.

One of our Proof of Concepts shows that the read command can be used to figure out what the user is currently listening to. It's also possible to gain code execution by overwriting functions that are cached in RAM using the write command. However, this requires carefully crafted injected code with techniques like trampolines to not interfere with the device's normal operation.

The `READ_ADDRESS` command looks as follows:

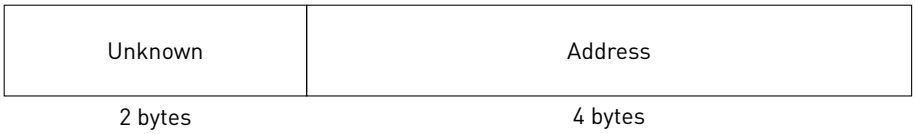

Figure 10: RACE READ_ADDRESS Command

The command will read 4 bytes from the given address. The purpose of the first two bytes is currently unknown, but setting them to null bytes (`0x0000`) works.

4.3.4 GET_BD_ADDRESS

This command retrieves the device's Bluetooth Device Address. This is helpful in situations where the attacker is able to connect to the device via BLE, but wants to connect via Bluetooth Classic. Impersonating the device also requires the knowledge of the device address. The command does not take any arguments, so the data part of the RACE message is empty.

The Bluetooth Device Address request only has one byte of payload. For TWS earbuds, this determines which device the request is targeted towards. For other audio devices this value is kept to zero.

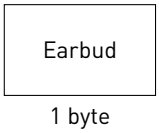

Figure 11: RACE GET_BD_ADDRESS Command

The payload of the device address response looks as follows:

Status Code	Earbud	Bluetooth Device Address
1 byte	1 bytes	6 bytes

Figure 12: RACE Get Bluetooth Device Address Response

The first byte is a status code, indicating whether the command was successful. The second byte indicates which earbud the response came from. At the end, the six byte Bluetooth Device Address follows.

4.4 RACE Transports

As mentioned above, RACE was initially discovered as a USB HID-based protocol. However, the reverse-engineering efforts uncovered that RACE can also be used via RFCOMM over Bluetooth Classic, or via GATT over BLE. There might be additional transports, but these are the ones that were observed in actual products.

For the Bluetooth based transports, the following illustration shows where the RACE protocol resides on the Bluetooth stack:

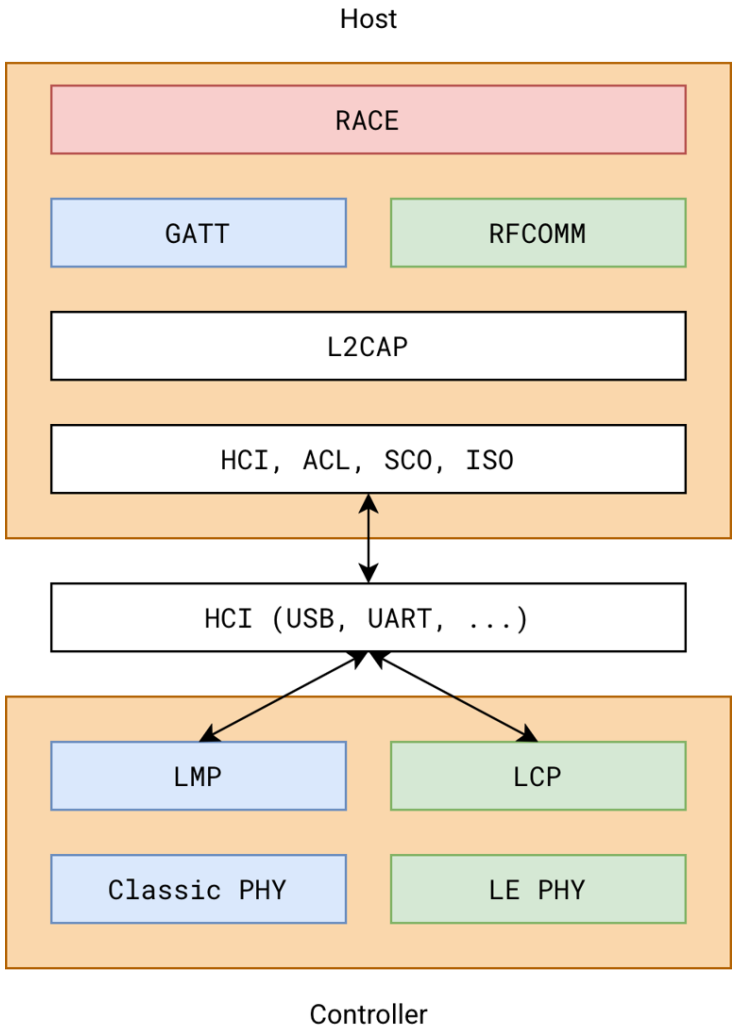

Figure 13: RACE in the Bluetooth Stack

4.4.1 USB Transport

The USB transport was the first one that was identified. Capturing the USB traffic revealed that there is a large amount of USB HID communication. The survey on actual devices showed only two devices that expose RACE via USB. The Bluetooth Auracast transmitter and the headphone model by the same vendor. All other headphones and earbuds either did not have a USB interface, or did not expose RACE via USB as the USB port was only used for charging.

In the USB HID based RACE transport, the USB HID header is required. The header consists of the HID report number (`0x06`) and the length of the following payload.

For RACE requests, the report number is `0x06`, for RACE responses the report number is `0x07`.

The format is shown below:

Report Number	Length	RACE Payload
1 byte	2 bytes	*Length* bytes

Figure 14: USB HID Transport

4.4.2 RFCOMM Transport

For Bluetooth Classic connections an RFCOMM channel is used to speak the RACE protocol. RFCOMM channels are usually identified via their UUIDs. In this case, various vendors have different vendor-specific UUIDs for their RACE RFCOMM channels. Vendors can choose custom UUIDs and set them via the device configuration that resides in the firmware.

The table below shows the different vendor UUIDs that were identified during the research:

Name	UUID
Common RFCOMM UUID	`00001101-0000-1000-8000-00805F9B34FB`
Airoha RFCOMM UUID	`00000000-0000-0000-0099-AABBCCDDEEFF`
Sony RFCOMM UUID	`8901DFA8-5C7E-4D8F-9F0C-C2B70683F5F0`
Bose RFCOMM UUID	`2D064AA9-32B5-4970-865C-643742BD2862`
Beyerdynamic RFCOMM UUID	`7B46E8DE-5A7E-4512-BE0D-863AEAA3312B`

After establishing a Bluetooth Classic connection and opening the respective RFCOMM channel, it can be used to speak the RACE protocol. On Linux, for example, this can easily be done using the `rfcomm` utility[1]. In most of the devices, the channel ID for the RFCOMM channel is `21`, regardless of the UUID. The following command will establish a connection to the channel and allow the user to speak the RACE protocol:

```
1 rfcomm bind $BDADDR 21
```

Listing 1: Rfcomm Utility Linux

The `rfcomm` utility will create a TTY device that can now be used like any other serial device.

[1] *https://linux.die.net/man/1/rfcomm*

4.4.3 GATT Transport

Via Bluetooth Low Energy, the devices expose a GATT service with two characteristics for subscribing and writing.

The table below lists the GATT service and characteristic UUIDs. In this case, most devices use the default Airoha UUIDs. Only Sony seem to be employing their own custom GATT UUIDs.

Name	UUID
Airoha GATT Service UUID	`5052494D-2DAB-0341-6972-6F6861424C45`
Airoha GATT TX UUID	`43484152-2DAB-3241-6972-6F6861424C45`
Airoha GATT RX UUID	`43484152-2DAB-3141-6972-6F6861424C45`
Sony GATT Service UUID	`DC405470-A351-4A59-97D8-2E2E3B207FBB`
Sony GATT TX UUID	`BFD869FA-A3F2-4C2F-BCFF-3EB1EC80CEAD`
Sony GATT RX UUID	`2A6B6575-FAF6-418C-923F-CCD63A56D955`

RACE commands are sent to the device by writing to the TX characteristic. RACE responses can be received by subscribing to the RX characteristic. If requests or responses are longer than the negotiated MTU, they need to be reassembled. Fragmentation and assembly do not introduce any additional overhead. The packets are cut off at the point where they exceed the MTU. The receiving side is responsible for keeping track of package size and reassembly, no further fragmentation indications and headers are added by the sender.

5 Vulnerabilities

This section will give an overview over the vulnerabilities and potential attack vectors that were identified during this research. In essence, the issues can be summarized as missing authentication in combination with a powerful debug-like protocol. This allows unauthenticated attackers to gain control over RAM and flash contents of the device.

This section will cover the authentication issues first and then briefly summarize the capabilities of the RACE protocol. Following that, the impact of the combination of the issues is outlined. The section ends with a discussion of the disclosure and patching process.

5.1 Authentication

5.1.1 Authentication for RACE

The RACE protocol itself does not implement any form of authentication. As such, it is entirely up to the interface or the transport that it is communicated over. Given its capabilities, RACE poses a threat if an attacker or any unauthorized party can communicate the protocol.

In most cases, none of the transports employed an additional layer of authentication. Only a very small number of device vendors did employ an authentication on Bluetooth level at the time of the discovery of this vulnerability.

An attacker within Bluetooth range can hence connect to the device and use the RACE protocol. An attacker with physical access can connect to devices that expose RACE via USB and use the RACE protocol. Both without any form of authentication.

5.1.2 Bluetooth Authentication

The lack of authentication on the Bluetooth layer is an issue in itself, regardless of the presence of the RACE protocol. The Airoha SDK does not seem to enforce pairing when a new Bluetooth connection is established. This means that an attacker within Bluetooth range can connect to such a target without any authentication or user interaction. This applies to both Bluetooth Classic and BLE.

One possible scenario is that an attacker can connect to a pair of headphones and sniff and record the microphone. However, the implications largely depend on the scenario and the target device itself. If the device only allows one Bluetooth connection at a time, and the owner is currently listening to music, they will notice the connection drop. If the device accepts multiple connections, the original audio connection is still likely to drop when the microphone is enabled on the attacker connection.

Depending on the functionalities of the target device, unauthenticated Bluetooth connections might give the attacker other capabilities.

Airoha has issued two separate CVEs for the Bluetooth authentication issue. One for Bluetooth Classic, and one for BLE:

- CVE-2025-20700: Missing Authentication for GATT Services
- CVE-2025-20701: Missing Authentication for Bluetooth BR/EDR

5.2 RACE Protocol Capabilities

The RACE protocol has been thoroughly described in Section 4. This section aims to summarize what an attacker could achieve once they are able to speak the RACE protocol.

The protocol offers a number of commands that grant extensive access to the devices. Most importantly all volatile and non-volatile storage can be read from and written to. This implies that any configuration or runtime data can be extracted or manipulated, including the code running on the devices. These four memory primitives, strictly speaking, imply all other capabilities, as those could be built using them. However, the RACE protocol offers convenient higher level commands. One example are the commands to read and write the so called *NVDM* partition. This partition holds the devices persistent configuration, which could be interfaced by reading and writing the raw flash data. But much more conveniently the configuration values can be read and modified using dedicated *NVDM* RACE commands.

Airoha has issued a CVE for the presence of critical capabilities within the RACE protocol:

- CVE-2025-20702: Critical capabilities of the RACE protocol

5.3 Impact

The later parts of this chapter will discuss the impact from a headphone user's perspective in more detail. First, the individual vulnerabilities are described with regard to the capabilities they provide an attacker.

1. CVE-2025-20700: Missing Authentication for GATT (BLE)

An attacker can discover and connect to headphones in proximity via BLE, e.g., on a train or in a café. Connection establishment usually happens silently without the user noticing. The lack of authentication provides covert access to the RACE protocol if it is exposed via GATT.

2. CVE-2025-20701: Missing Authentication for Bluetooth Classic (BR/EDR)

Similar to the BLE flaw, this allows unauthorized connections. While Bluetooth Classic connections are sometimes more visible (and might interrupt audio streams), this vector is interesting even without the RACE protocol, as the

attacker can still establish two-way audio connections to the headphones or use any other Bluetooth profile that the headphones expose.

3. CVE-2025-20702: Critical Capabilities in Custom Protocol

The RACE protocol itself, accessed with or without relying on the previous two CVEs provides powerful commands intended for factory debugging. It allows reading and writing to the device's Flash memory and RAM. It is therefore possible to permanently alter devices and extract sensitive configuration data.

5.3.1 Attacking the Headphones

To compromise a pair of headphones, an attacker first needs to connect to them. Taking advantage of CVE-2025-20700, for the initial connection is considerably easier than using CVE-2025-20701. The reason for this is the fact that headphones typically advertise their presence via BLE. Anyone in range can scan for BLE devices, connect to them and use the appropriate GATT service to speak the RACE protocol. In contrast, devices will not typically announce their presence or respond to inquires on Bluetooth Classic, except when they are explicitly set to discoverable or in pairing mode. To establish a connection the device's Bluetooth Classic address has to be known. Finding the address of headphones in proximity to the attacker requires special purpose hardware, such as the Ubertooth[12]. It is however worth noting, that for many affected devices the Bluetooth Classic address can be inquired via RACE. This means attackers that want to establish a Bluetooth Classic connection can simply connect via BLE first and use RACE to find out the Classic address.

5.3.1.1 Eavesdropping

Using Bluetooth Classic and the Bluetooth Hands-Free Profile (HFP) attackers can use the headphone's microphone. This is a direct consequence of the broken or missing authentication and does not rely on the RACE protocol's capabilities. As noted earlier, establishing a Bluetooth Classic audio connection usually disrupts any pre-existing connection. In case the victim is actively listening to audio they will notice that something is happening to their headphones.

5.3.2 Data Extraction

As mentioned earlier, volatile (RAM) and non-volatile (flash) storage can be read using RACE. Reading RAM allows an attacker to extract runtime information of different, sensitive natures. As a proof of concept, we demonstrated that we can read out metadata related to the currently playing media. This means it is possible to find out what the user is currently listening to.

```
❯ uv run race_toolkit.py mediainfo
INFO: Trying to dump current playing media info. Identifying model and firmware version first...
INFO: Device initialized.
INFO: Scanning for BLE devices...
INFO: Found 4 matching devices:
INFO: [0]: [redacted]
INFO: [1]: [redacted]
INFO: [2]: WH-CH720N (88:92:[redacted]/P)
INFO: [3]: [redacted]
Which one do you want to connect to?
2
INFO: Negotiated GATT MTU to 242.
INFO: Found the Sony RACE UUID 5052494D-2DAB-0341-6972-6F6861424C45!
INFO: Got buildversion `mt2822x_evkMT2822_SDK_Sony-ER69_mdr14_c42sp_12024/09/18 18:58:55 GMT +08:00`.
INFO: Your target is currently listening to:
INFO:   Track: Look What You Made Me Do
INFO:   Album: Taylor Swift
INFO:   Artist: reputation
INFO:   Genre: Pop
```

Figure 15: RACE Toolkit Mediainfo Command

Access to flash data allows extracting the firmware and any other data stored on the flash. This includes information on the devices the headphones have been paired with. This is an important prerequisite for other attacks, that will be discussed in the next section.

5.3.2.1 Arbitrary Code Execution

Writing to RAM or flash sections that hold executable code permits attackers to run any custom code on the device. Temporarily injecting code into RAM is of questionable use, considering the capabilities attackers have even without doing that.

Permanently infecting headphones by writing code to their flash could however be interesting to attackers. However, this process is much more complex. The flash write command restricts the areas that can be written to. This restricted flash area is part of the firmware update process. Writing is only possible in the firmware update partition. Once the firmware is written to the partition, the update process has to be triggered. This requires the ability to create a valid firmware image. However, with the given flash read capabilities, the required symmetric keys can be read and used to encrypt a firmware image. In practice, this process takes some time. Depending on the size of the firmware image, we observed that flashing the image via RFCOMM takes about 3 to 5 minutes. Via GATT the same process likely takes even longer.

5.3.3 Attacking the Phone

Up to this point, the damage is confined to the headphones themselves. That is already undesirable, but it gets worse once a more interesting target, such as the user's phone gets involved. When the three vulnerabilities are chained together, the impact can be shifted from "hacking headphones" to "compromising the phone."

To carry out the attack the following steps have to be performed:

Step 1: Connect (CVE-20700/20701)

The attacker is in physical proximity and silently connects to a pair of headphones via BLE or Classic Bluetooth.

Step 2: Exfiltrate (CVE-20702)

Using the unauthenticated connection, the attacker uses the RACE protocol to (partially) dump the flash memory of the headphones.

Step 3: Extract

Inside that memory dump resides a connection table. This table includes the names and addresses of paired devices. More importantly, it also contains the Bluetooth Link Key. This is the cryptographic secret that a phone and headphones use to recognize and trust each other.

Note: Once the attacker has this key, they no longer need access to the headphones.

Step 4: Impersonate

The attacker's device now connects to the targets phone, pretending to be the trusted headphones. This involves spoofing the headphones Bluetooth address and using the extracted Link Key.

Once connected to the phone the attacker can proceed to interact with it from the privileged position of a trusted peripheral.

Many of the proof of concepts that are targeted at the phone require a proper implementation of Bluetooth profiles, such as the Hands-Free Profile (HFP). For these attacks, the tooling was based on BTstack[4], which is a great implementation of the Bluetooth host stack [2].

5.3.3.1 Data Extraction

Using HFP commands, the attacker can access different types of information. Some are restricted by the permissions the user gave during initial pairing to the headphones, some are always available.

[2] *This is probably the only project where you can run `git clone && make`, and it just works. Exactly what we needed here. If you ever need a Bluetooth host stack, we can really recommend BTstack!*

Using the HFP `AT+CNUM`, the attacker can obtain the victim's phone number. If access to the phone book was granted during initial pairing, the attacker can also access contact lists and call history.

The following screenshot shows the output of a tool based on the BTstack HFP example[3]. A connection to a victim phone is established and the `AT+CNUM` command is sent. The device responds with the phone number.

```
❱ ./hfp_hf_console A0:[redacted]
Packet Log: /tmp/hci_dump.pklg
Got target device address parameter. Target address is: A0:[redacted]
SCO Demo: Sending and receiving audio via btstack_audio.
USB device 0x0a12/0x0001, path: 01
Local version information:
- HCI Version    0x0006
- HCI Revision   0x22bb
- LMP Version    0x0006
- LMP Subversion 0x22bb
- Manufacturer   0x000a
TLV path: /tmp/btstack_88-[redacted]
BTstack up and running on 88:[redacted].
Supported codecs: CVSD, mSBC
Establish Service level connection to device with Bluetooth address A0:[redacted]...
Service level connection established A0:[redacted].

AG Indicator Mapping | INDEX 1: range [0, 1], name 'service'
AG Indicator Mapping | INDEX 2: range [0, 1], name 'call'
AG Indicator Mapping | INDEX 3: range [0, 3], name 'callsetup'
AG Indicator Mapping | INDEX 4: range [0, 5], name 'battchg'
AG Indicator Mapping | INDEX 5: range [0, 5], name 'signal'
AG Indicator Mapping | INDEX 6: range [0, 1], name 'roam'
AG Indicator Mapping | INDEX 7: range [0, 2], name 'callheld'
AG Indicator Status  | INDEX 1: status 0x01, 'service'
AG Indicator Status  | INDEX 2: status 0x00, 'call'
AG Indicator Status  | INDEX 3: status 0x00, 'callsetup'
AG Indicator Status  | INDEX 4: status 0x03, 'battchg'
AG Indicator Status  | INDEX 5: status 0x02, 'signal'
AG Indicator Status  | INDEX 6: status 0x00, 'roam'
AG Indicator Status  | INDEX 7: status 0x00, 'callheld'
Speaker volume: gain 9
Microphone volume: gain 9
Apple Extension supported: yes
Query Subscriber Number
Subscriber number: +4917[redacted]
```

Figure 16: HFP AT CNUM Command

5.3.3.2 Remote Control

Using the HFP `AT+BVRA` command, the attacker can trigger voice assistants (e.g. Siri or Google Assistant). Depending on the settings on the phone, this can be used to send text messages, make calls, or perform other actions on the phone.

On iOS, Siri is enabled on the lock screen by default. However, in the locked state the possible actions that can be performed are restricted. It is possible to send text messages or make calls. Other information, such as location information, or access to notes is not possible while the phone is locked.

5.3.3.3 Hijacking Calls

Using the calling features of the Hands-Free Protocol, an attacker can make calls or accept calls. HFP also offer various other possible actions on phone calls, such as merging calls, holding calls, or obtaining the call status.

This allows an attacker to silently accept a call and receive the audio stream of the call. Alternatively, an attacker could call expensive premium numbers to cause financial damage.

5.3.3.4 Eavesdropping

In addition to the previous attacks, the phone capabilities also allow for eavesdropping. The attacker can trigger a call to an attacker-controlled phone number. Once the call is established, the Bluetooth audio connection is dropped.

In consequence there will be an active phone call to the attacker using the victim phone's internal microphone.

5.3.4 Other Hosts

Smartphones are certainly the most common host devices for Bluetooth headphones and the attacks that have been described so far have were sketched out with that in mind. However, other host systems can also fall victim to the above impersonation attack. The specific impact varies greatly. Most telephony related attacks are typically not possible for desktop computers. Voice assistants are however potential targets.

5.4 Vulnerable Devices

During this research we conducted a survey to identify what devices are affected by the vulnerabilities. In a first instance, this requires identifying devices with Airoha Bluetooth SoCs. This is a task that can be done by passive reconnaissance. For example, by examining FCC reports which usually contain photographs of the PCBs inside the device. In such cases, it might be possible to identify the SoC. Other sources can be tear-down or disassembly blogs. One notable example is the blog 52audio.com[13], which provides high-resolution tear-down pictures of many audio devices. Searching for Airoha SoCs on their page revealed many potential candidates for the vulnerabilities. Lastly, members of the Bluetooth

Special Interest Group (SIG) have the possibility to search for devices with specific Bluetooth SoCs. However, ERNW is not a member, as such this search was not available to us.

Once potential vulnerable devices are identified, the more difficult step is to confirm whether they are in fact vulnerable. For this purpose, we acquired several devices. However, it was not possible for us to buy all potential devices. As such, only a subset of devices could be confirmed to be affected.

During our survey, some Airoha-based devices that were not vulnerable to the Bluetooth Classic vulnerability (CVE-2025-20701) were found. A notable example here are the Jabra Elite 8 Active earbuds. The assumption is that these vendors adapted the SDK code to enforce pairing, or changed default configurations of the SDK to secure the device. For some models we found that the pairing was unreliable. Sometimes, the device did not accept HFP or RACE connections without prior pairing, sometimes it did. An example for such a device are the Bose QuietComfort Earbuds.

Not every device exposed RACE over all described transports. However, this is likely because the interface itself was not exposed. For example, the Bose QuietComfort Earbuds or the Jabra Elite 8 Active do not seem to have BLE enabled, which effectively means that GATT is not available.

In the initial partial disclosure the following list of vulnerable devices was published. This list is not a complete list of affected devices. It only includes devices that we managed to verify. For this purpose, we acquired some of the devices, or asked colleagues and friends whether we could check their devices. For some devices, it was only possible to verify the vulnerability via BLE, as this check can be conducted with a smartphone and does not necessarily require a computer.

- Beyerdynamic Amiron 300
- Bose QuietComfort Earbuds [3]
- EarisMax Bluetooth Auracast Sender
- Jabra Elite 8 Active [4]
- JBL Endurance Race 2
- JBL Live Buds 3
- Jlab Epic Air Sport ANC
- Marshall ACTON III
- Marshall MAJOR V
- Marshall MINOR IV
- Marshall MOTIF II
- Marshall STANMORE III
- Marshall WOBURN III
- MoerLabs EchoBeatz
- Sony Link Buds S
- Sony ULT Wear
- Sony WF-1000XM3
- Sony WF-1000XM4
- Sony WF-1000XM5
- Sony WF-C500
- Sony WF-C510-GFP
- Sony WH-1000XM4
- Sony WH-1000XM5
- Sony WH-1000XM6
- Sony WH-CH520
- Sony WH-CH720N

[3] *No GATT exposed. Classic pairing vulnerable, but unreliably.*
[4] *No GATT exposed, Classic pairing is enforced, not vulnerable.*

- Sony WH-XB910N
- Sony WI-C100
- Teufel Tatws2

Due to the sheer amount of devices that are potentially still affected, there is no proper overview over the current status of fixes

There a few vendors that mention in firmware updates that the issues are fixed. Other vendors might have silently patched, some might not have patched the vulnerability at all. A positive examples is Jabra, who mentioned the CVEs in firmware release notes[11] as well as their security center overview[10]. Which is interesting, because they are less affected than some other vendors due to additional security measures. Moreover, they list all devices that are affected, not just the one device that were initially reported.

Below is a short list of vendor security updates that are known to us:

- **Jabra:** *https://www.jabra.com/support/release-notes/release-note-jabra-elite-8-active*
- **Marshall:** *https://www.marshall.com/en/en/support/speakers/support-for-acton-iii/latest-firmware-version*
- **Beyerdynamic** (no public link, update supplied via mobile app)

This large field of unclear information is another motivation for publicly disclosing the technical details. They should empower users to check their own devices. The initial plan was to create and maintain a list of devices. However, acquiring, updating, and testing all these devices would take up time and resources. In the end, it's the manufacturers' responsibility to patch and inform their users.

5.5 Mitigation

5.5.1 Users

The most important advice for users is to update their devices and keep them updated. However, a firmware update might not be available, or the vendor might not have released information about whether the vulnerabilities were addressed in an update. Users that are unsure about whether their device is still affected can try the verification methods shown in Section 6. However, it is important to note that these methods cannot guarantee the absence of the vulnerabilities.

Individuals that consider themselves high risky targets, such as journalists, diplomats, politicians, or any other persons threatened by surveillance are advised to use wired headphones instead of Bluetooth headphones.

Additionally, it is advisable to regularly monitor the list of paired devices on the phone. If there are a lot of old, unused ones, they should be removed. That's one less Link Key per device that is potentially stolen.

5.5.2 Manufacturers

Manufacturers should apply the Airoha SDK patches as soon as possible. Ideally, this should have happened as soon as Airoha released the SDK update with the information about the vulnerability.

Moreover, it is highly recommended for manufacturers to do security assessments before releasing the products. Following Bluetooth security testing methodology, such as BSAM[16] could have caught these issues.

5.5.3 The Vulnerabilities Themselves

Very simplified and summarized, the vulnerabilities can be considered as missing authentication and exposed debug functionality.

1. CVE-2025-20700: Missing Authentication for GATT (BLE)

The GATT service should not be available to unpaired devices. GATT offers the functionality to require authentication for characteristics. There are other characteristics and services that are required to be available without pairing (e.g., the Google Fast Pair services), so a service-based authentication for GATT is sensible.

2. CVE-2025-20701: Missing Authentication for Bluetooth Classic (BR/EDR)

An unauthenticated Bluetooth Classic connection should never be possible. The device should only accept new connections when it is set in pairing mode.

3. CVE-2025-20702: Critical Capabilities in Custom Protocol

The RACE protocol should not expose critical functionality, such as reading and writing RAM or flash.

5.6 Disclosure

Below is the detailed disclosure timeline. We want to emphasize that, once the technical Email related issues were fixed, the communication with Airoha was very positive. Airoha was committed to fix the issues and provide their customers with an updated SDK quickly. We also contacted three manufacturers. Sony and Marshall did not respond to our Emails. Sony only started to respond to our inquiries once they learned that a public partial disclosure and presentation of the vulnerabilities was planned at the Troopers conference. Beyerdynamic, driven by the upcoming EU Cyber Resilience Act (CRA), responded to our Emails which lead to multiple constructive exchanges between ERNW and Beyerdynamic.

Sony offers a bug bounty program via HackerOne. However, it was not possible to submit the issue via the platform. The terms and conditions that are required by the program are not suitable for this vulnerability. ERNW would renounce all rights to the vulnerability. As such, we would not have been allowed to share the vulnerability with Airoha or other manufacturers. Moreover, all rights to disclose details would be taken away from us.

5.6.1 Disclosure Timeline

Date	Description
March 25, 2025	Initial notification and report to security@airoha.com. Start of 90-day disclosure period.
March 26, 2025	ERNW provides detailed vulnerability report to Airoha.
March 31, 2025	ERNW polls Airoha via another Email.
April 07, 2025	ERNW polls Airoha via Email and website contact form.
April 14, 2025	ERNW polls Airoha via another Email and announces that it will contact the BSI and some of the vendors by April 17 3pm German time.
April 24, 2025	ERNW contacts Sony (security@sony.com), Marshall (soc@marshall.com), Beyerdynamic (info@beyerdynamic.de) via Email (one week later than announced).
April 24, 2025	Marshall: Automatic response to confirm reception of Email.
April 28, 2025	Beyerdynamic: Compliance Manager responds with request for call.
April 30, 2025	Beyerdynamic: Call with short summary of the vulnerabilities and coordination of further process.
May 05, 2025	Marshall: ERNW polls. Automatic response received.
May 07, 2025	Beyerdynamic: ERNW sends detailed report with vendor-specific PoCs.
May 08, 2025	Beyerdynamic: Confirms reception of report and asks for meeting.
May 09, 2025	Airoha: ERNW polls Airoha via Email.
May 09, 2025	Sony: Polling Sony via various Email addresses with explanation why HackerOne is not possible.
May 09, 2025	Sony: Mail to privacyoffice.SEU@sony.com due to potential privacy issues with reference to EU CRA.
May 12, 2025	Sony: Private contact to Sony tries to find out what Email we can contact.
May 13, 2025	Sony: Email to psirt@sony.com
May 16, 2025	Sony: Response from productsec-head@jp.sony.com. Request to accept terms that turned out to be the same ones as HackeOne. ERNW responds that this is not possible due to multi-vendor disclosure and asks on how to proceed.
May 16, 2025	Marshall: ERNW polls. Automatic response received.
May 16, 2025	Airoha: ERNW polls Airoha via Email.
May 21, 2025	Beyerdynamic: Meeting to discuss findings.
May 27, 2025	Airoha: First response to ERNW. Communication begins, Airoha is very committed to handling the situation well.
June 04, 2025	Airoha: Supplies vendors an SDK update with mitigations.

Date	Description
June 26, 2025	Partial Disclosure of the vulnerabilities by ERNW via a blog post[9] and presentation at TROOPERS[6][5].
Dec 27, 2025	Full Disclosure of the vulnerabilities by ERNW at 39c3[5].

[5] *https://troopers.de/*

6 Tools

In this section we briefly outline the tooling we released alongside this white paper. Moreover, we will provide instructions for both technical and non-technical users to verify whether their devices are affected. To be certain that a device is affected, we recommend using our RACE Toolkit, as the non-technical approach does have technical limitations, such as only verifying the availability of the RACE protocol via BLE.

Neither method can absolutely guarantee the absence of vulnerabilities. They are only provided to support the assessment of a device and result in an estimation whether a device might not be vulnerable to the described issues any longer.

6.1 For Non-Technical Users

A simple method to evaluate whether the RACE protocol is exposed via BLE on a device is by using the *nRF Connect* mobile app. It is available for both iOS and Android in their respective App Stores.

- Android: https://play.google.com/store/apps/details?id=no.nordicsemi.android.mcp
- iOS: https://apps.apple.com/de/app/nrf-connect-for-mobile/id1054362403

The following instructions should work for any Airoha based headphone or earbud model that has BLE enabled. An example of a device that is Airoha based, but does not expose any services via BLE are the Bose QuietComfort Earbuds or the Jabra Elite 8 Active. For these kind of devices, we suggest using the RACE Toolkit (see Section 6.2).

One other exception are Sony devices. These devices use different UUIDs. Whenever this becomes relevant, we will mention the difference.

We will first demonstrate the evaluation method for iOS devices in Section 6.1.1 and then for Android devices in Section 6.1.2. The evaluation will consist of scanning and connecting to the target device, enumerating the GATT services and characteristics, and communicating with the device. As a first test, the Get Build Version command (see Section 4.3.1) is sent. There are a number of devices that do not, or will no longer respond to this command. As such, the second test will be using the Read Flash command (see Section 4.3.2) to check whether it is possible to read the flash contents of the device.

Ideally, the device will not allow a connection or disallow sending and subscribing to GATT characteristics.

6.1.1 nRF Connect Mobile on iOS

Step 1: Filtering

The first thing that should be done is setting a filter. There are usually many Bluetooth devices in the vicinity. Tapping the arrow button on the top right opens up the filter menu. Switch on *Remove Unnamed* and *Remove Unconnectable*. We just want to show devices that are actually connectable, and we want to be able to identify them by their name. If the target device was not renamed explicitly, it should be advertised under its model name (e.g. `WH-CH720N` in this example).

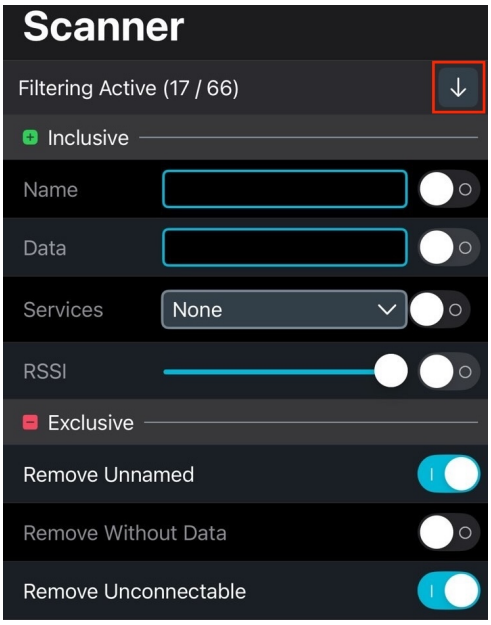

Figure 17: iOS Set Filter

Step 2: Connecting

Once the filter is active, you can look for the target device. In this example, we're connecting to the Sony WH-CH720N headphones. Tap the *Connect* button to establish a BLE connection. Some device models might request pairing at this point. Accept the pairing if necessary. If there's no prompt for a pin or button press, this is still not a proper authentication.

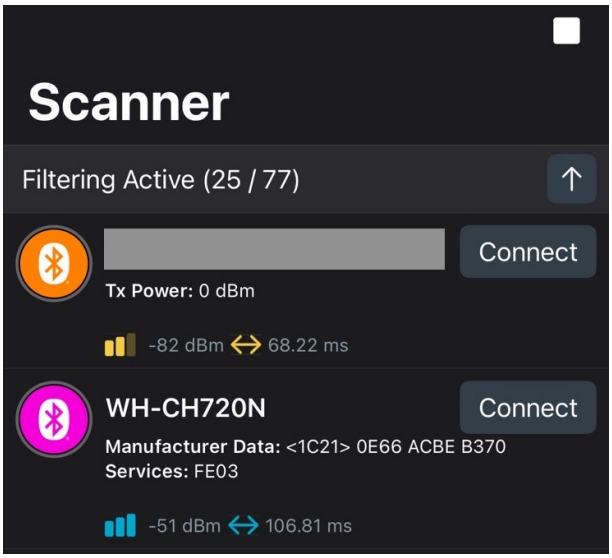

Figure 18: iOS Connect to Device

Once the connection is established, the app will show GATT services and characteristics. Make sure that you are on the second tab (highlighted in red).

If you are testing a Sony device, the GATT service UUID you are looking for is

`DC405470-A351-4A59-97D8-2E2E3B207FBB`.

If your device is by another vendor, you are looking for the default service UUID

`5052494D-2DAB-0341-6972-6F6861424C45`.

In this case, we're testing a Sony device.

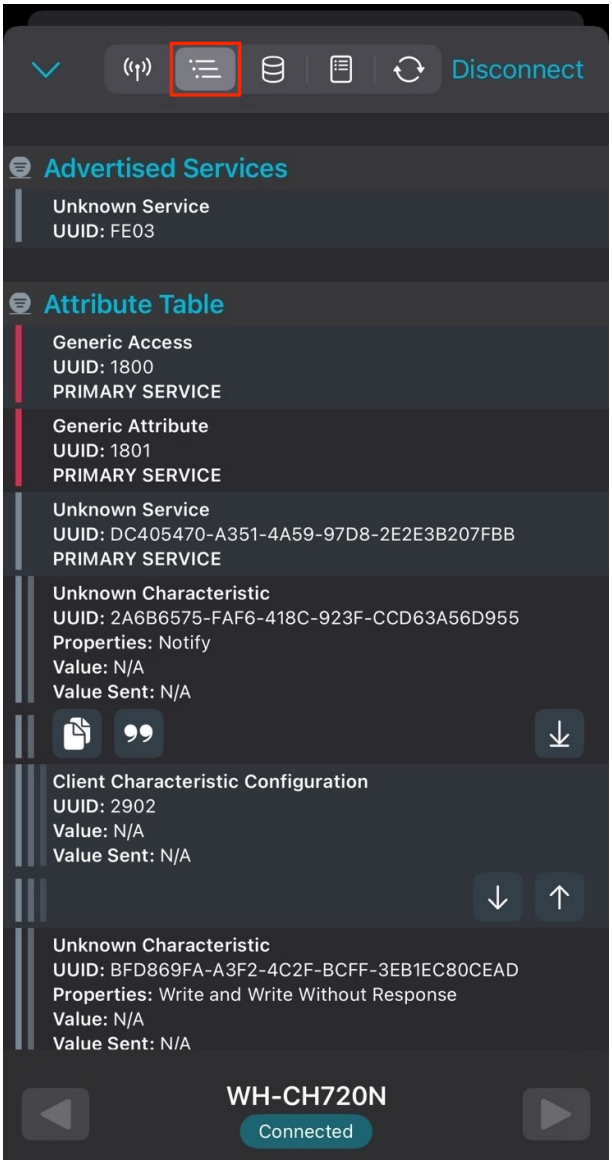

Figure 19: iOS GATT Services

Step 3: Subscribing to GATT Characteristic

Now we need to subscribe to the reception characteristic. This is the only subscribable characteristic within the GATT service. Subscribable characteristics are indicated by the arrow down buttons with a horizontal line (see red highlighting). Tap the button to subscribe to the characteristic. In case of Sony devices is it

`2A6B6575-FAF6-418C-923F-CCD63A56D955`,

for other devices the UUID is

`43484152-2DAB-3141-6972-6F6861424C45`.

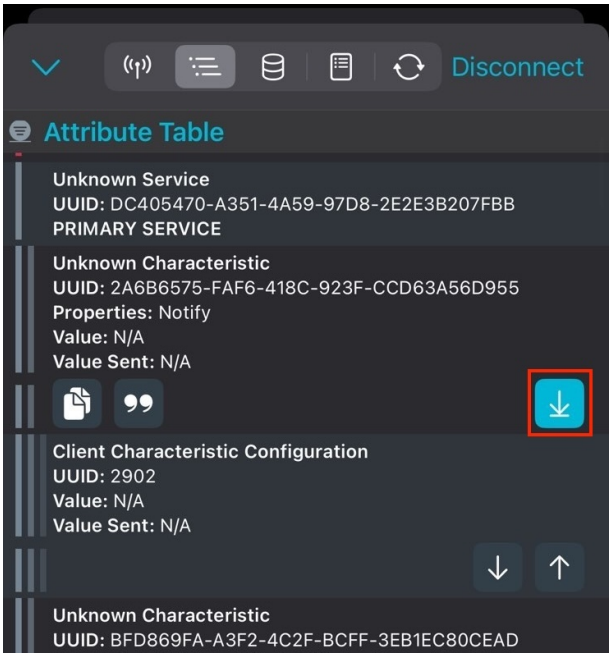

Figure 20: iOS Subscribe GATT

Step 4: Sending Get Build Version Command

After successfully subscribing to the reception characteristic. We can write into the transmission characteristics to send data to the device. For Sony devices this characteristic is

`BFD869FA-A3F2-4C2F-BCFF-3EB1EC80CEAD`,

for other devices the UUID is

`43484152-2DAB-3241-6972-6F6861424C45`.

Tapping the arrow up button opens the write window. First, we will send the Get Build Version command. The value `055A0200081E` can be copied and inserted into the window. Tap the write button to send the command to the device.

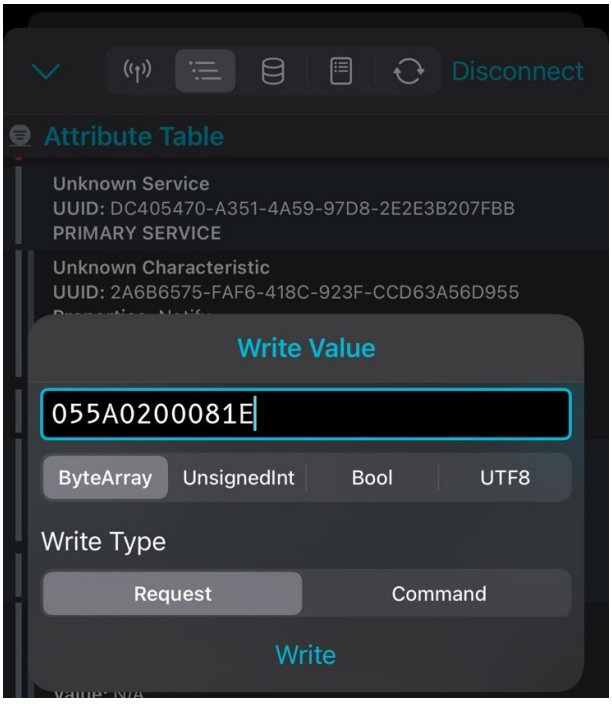

Figure 21: iOS Send Get Build Version

The response should be received soon after sending the command. It will appear in the subscribed reception characteristic as shown in the screenshot below. If no response is received, try sending the data again. If this still does not result in a response, the device might not respond to the command. Proceed with step 5.

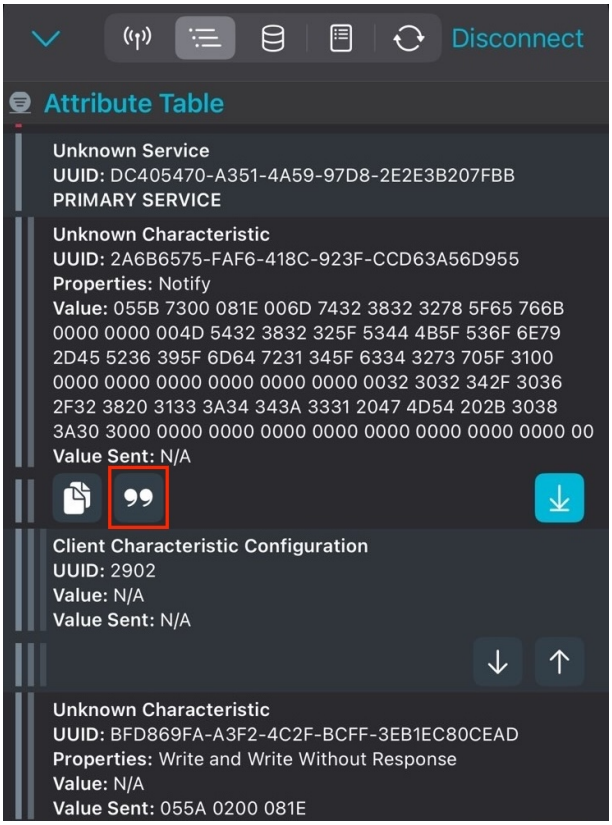

Figure 22: iOS Get Build Version Response

Tapping the highlighted button opens up the *Set Data Parser* window, which allows interpreting the received data as human-readable strings. Tap *UTF-8* and the window should close.

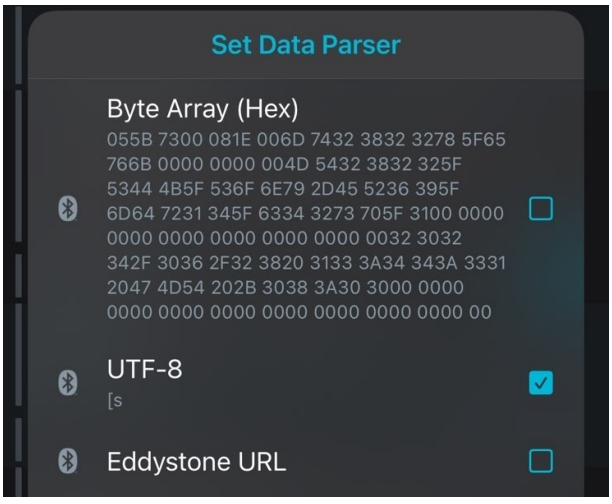

Figure 23: iOS Set Data Parser

As a result, the received data should now be a readable string. More information on the how to interpret this information can be found in Section 4.3.1.

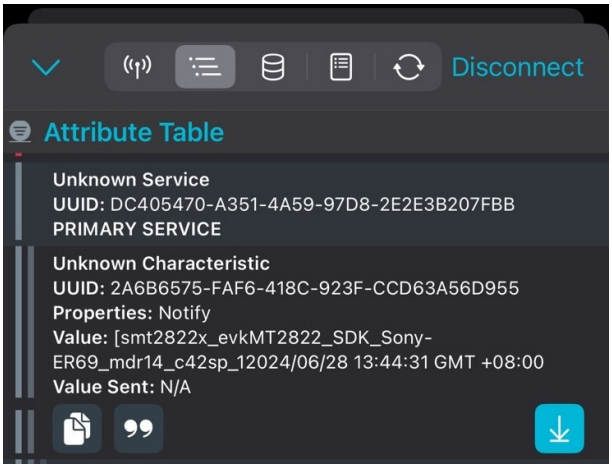

Figure 24: iOS Build Version UTF-8

Step 5: Sending Read Flash Command

Alternatively, or in addition to the Get Build Version command, the Read Flash command can be sent. If step 4 was executed, and the format changed to UTF-8, it should be set back to *Byte Array (Hex)* for this command.

Tap the arrow up button again on the transmission characteristic (`BFD869FA-A3F2-4C2F-BCFF-3EB1EC80CEAD` for Sony devices, `43484152-2DAB-3241-6972-6F6861424C45` for other devices). Copy the following data into the *Write Value* window:

`055A08000304000100001008`.

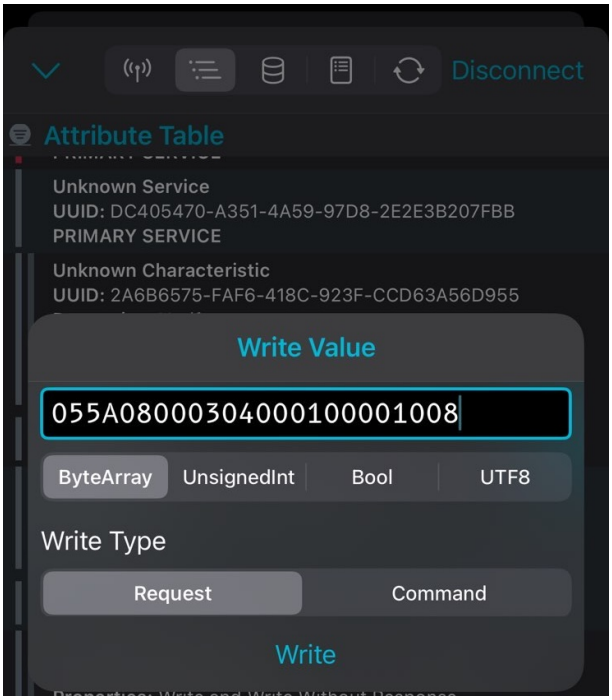

Figure 25: iOS Read Flash Command

After a short time, the response should be received in the reception characteristic. As the response is usually longer than what fits into a single GATT response, the data is fragmented and sent in several steps. The complete data can be viewed in the log. To open the log, tap the log icon (indicated in red).

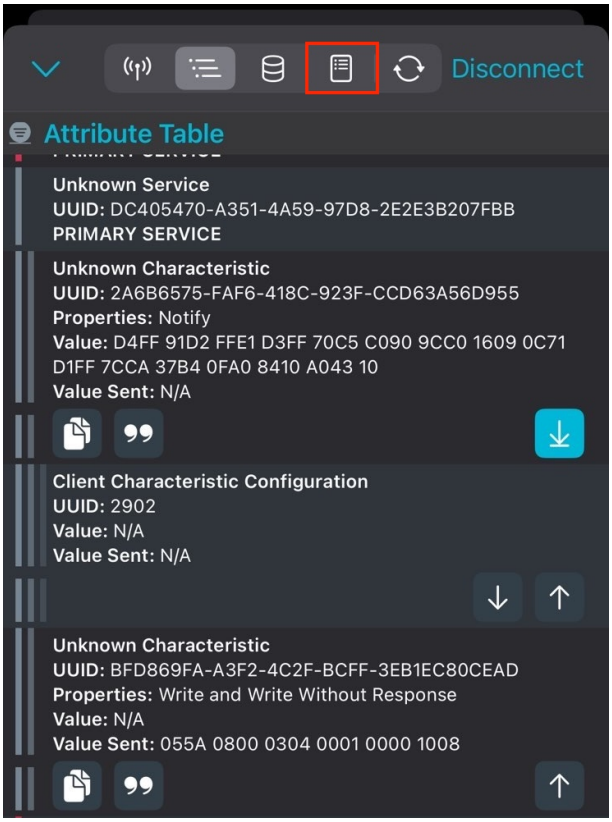

Figure 26: iOS Read Flash Response

A successful Read Flash response should look similar to the one below. The exact data is likely different. However, the response should be much longer than the command that was sent. A response with the length of the request might indicate an error. Check the beginning of the response. After the `03 04` there should be a NULL byte `00`. A `03` indicates an error, which means that the flash could not be read.

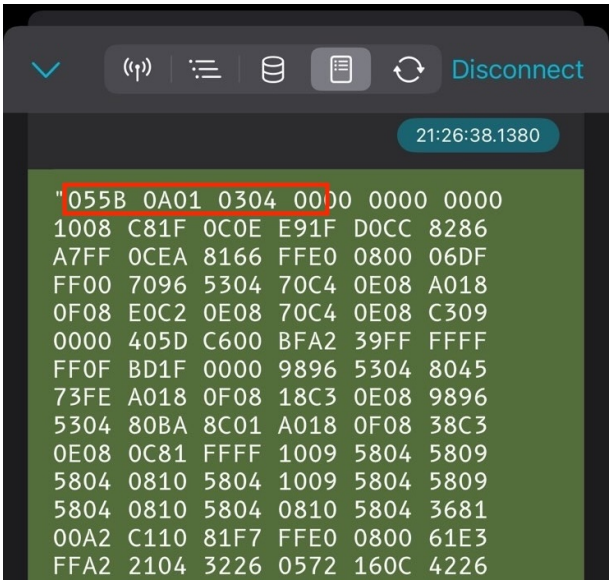

Figure 27: iOS Log

Conclusion

If these steps were successful, your device is vulnerable. An attacker can read out sensitive data from the device's flash and potentially use other RACE commands. Check whether the vendor of your device offers a firmware update.

6.1.2 nRF Connect Mobile on Android

Step 1: Filtering

The first thing that should be done is setting a filter. There are usually many Bluetooth devices in the vicinity. Tapping the tree dots under the menu bar opens the filter menu. Tap on *Type* and select *Connectable*. We just want to show devices that are actually connectable, and we want to be able to identify them by their name. If the target device was not renamed explicitly, it should be advertised under its model name (e.g. `WH-CH720N` in this example).

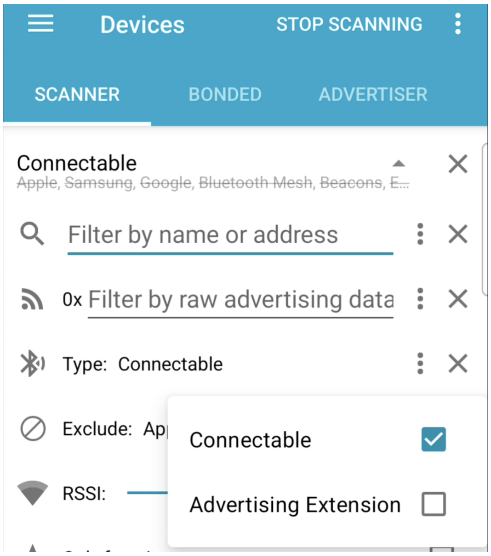

Figure 28: Android Set Filter

Step 2: Connecting

Once the filter is active, you can look for the target device. In this example, we're connecting to the Sony WH-CH720N headphones. Tap the *Connect* button to establish a BLE connection. Some device models might request pairing at this point. Accept the pairing if necessary. If there's no prompt for a pin or button press, this is still not a proper authentication.

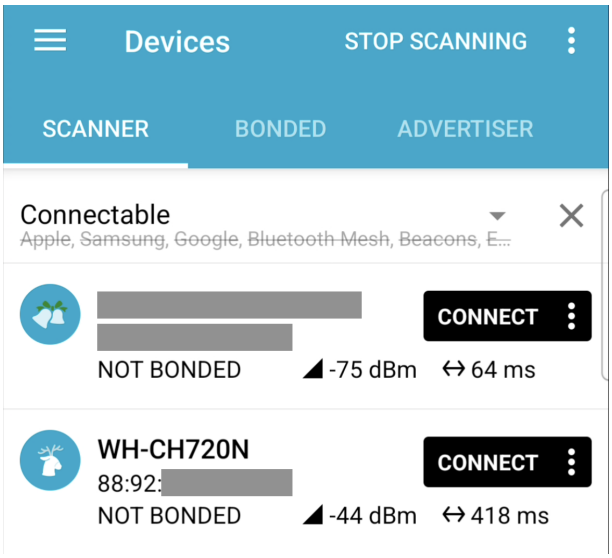

Figure 29: Android Connect to Device

Once the connection is established, the app will show GATT services and characteristics. Make sure that you are on the second tab (highlighted in red).

If you are testing a Sony device, the Service UUID you are looking for is

`DC405470-A351-4A59-97D8-2E2E3B207FBB`.

If your device is by another vendor, you are looking for the default service UUID

`5052494D-2DAB-0341-6972-6F6861424C45`.

In this case, we're testing a Sony device.

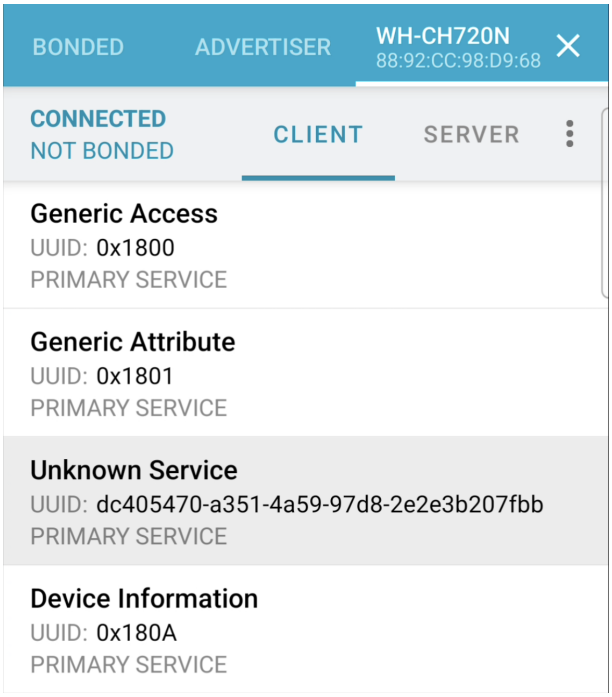

Figure 30: Android GATT Services

Step 3: Subscribing to GATT Characteristic

Now we need to subscribe to the reception characteristic. This is the only subscribable characteristic within the GATT service. Subscribable characteristics are indicated by the three arrow down buttons with a horizontal line (see red highlighting). Tap the button to subscribe to the characteristic. In case of Sony devices is it

`2A6B6575-FAF6-418C-923F-CCD63A56D955`,

for other devices the UUID is

`43484152-2DAB-3141-6972-6F6861424C45`.

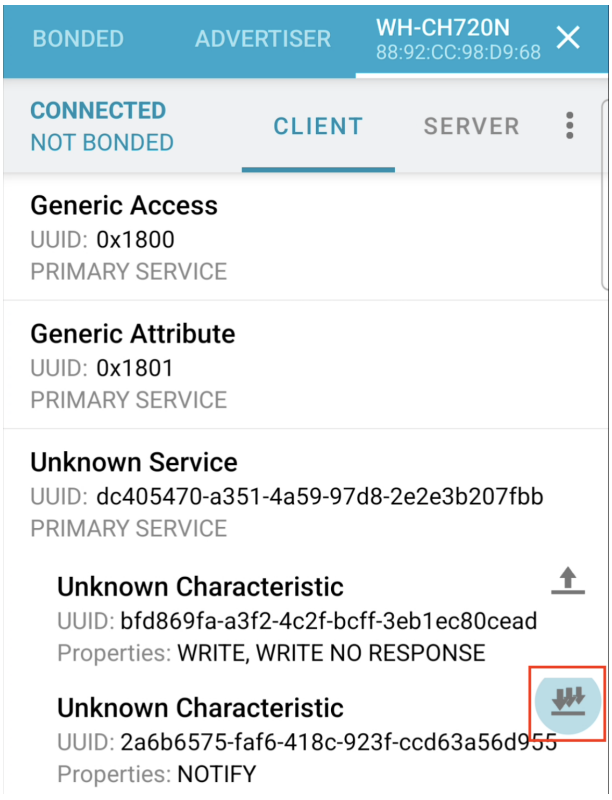

Figure 31: Android Subscribe GATT

Step 4: Sending Get Build Version Command

After successfully subscribing to the reception characteristic. We can write into the transmission characteristics to send data to the device. For Sony devices this characteristic is

`BFD869FA-A3F2-4C2F-BCFF-3EB1EC80CEAD`,

for other devices the UUID is

`43484152-2DAB-3241-6972-6F6861424C45`.

Tapping the arrow up button opens the write window. First, we will send the Get Build Version command. The value `055A0200081E` can be copied and inserted into the window. Tap the send button to send the command to the device.

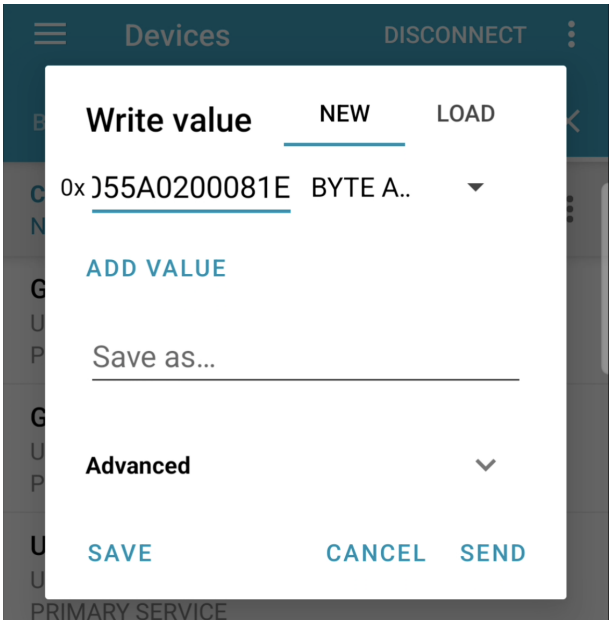

Figure 32: Android Send Get Build Version

The response should be received soon after sending the command. It will appear in the subscribed reception characteristic as shown in the screenshot below. If no response is received, try sending the data again. If this still does not result in a response, the device might not respond to the command. Proceed with step 5.

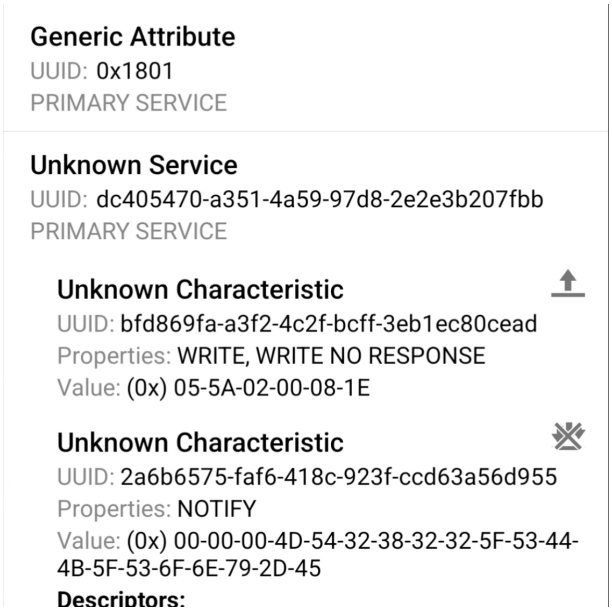

Figure 33: Android Get Build Version Response

Step 5: Sending Read Flash Command

Alternatively, or in addition to the Get Build Version command, the Read Flash command can be sent.

Tap the arrow up button again on the transmission characteristic (`BFD869FA-A3F2-4C2F-BCFF-3EB1EC80CEAD` for Sony devices, `43484152-2DAB-3241-6972-6F6861424C45` for other devices). Copy the following data into the *Write Value* window:

`055A08000304000100001008`.

This will read 256 bytes of the device's flash content at the address `0x08100000`.

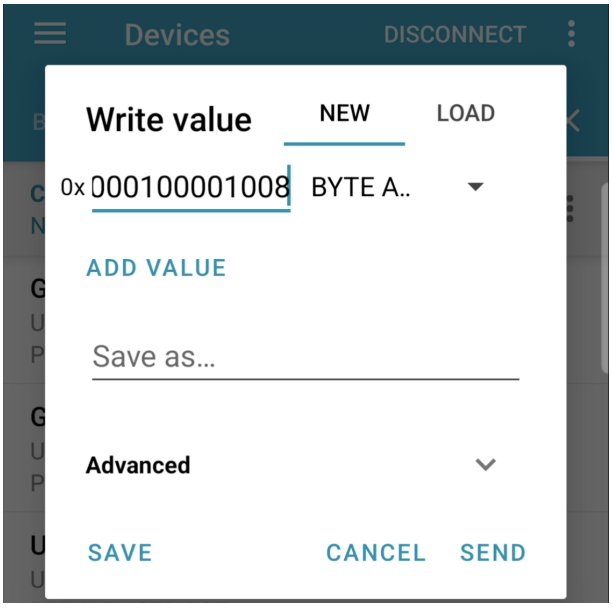

Figure 34: Android Read Flash Command

After a short time, the response should be received in the reception characteristic. As the response is usually longer than what fits into a single GATT response, the data is fragmented and sent in several steps. The complete data can be viewed in the log. To open the log, swipe to the right.

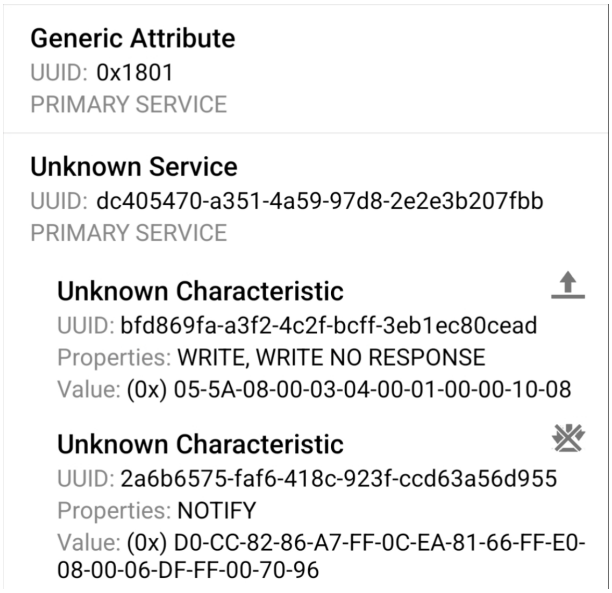

Figure 35: Android Read Flash Response

A successful Read Flash response should look similar to the one below. The exact data is likely different. However, the response should be much longer than the command that was sent. A response with the length of the request might indicate an error. Check the beginning of the response. After the `03 04` there should be a NULL byte `00`. A `03` indicates an error, which means that the flash could not be read.

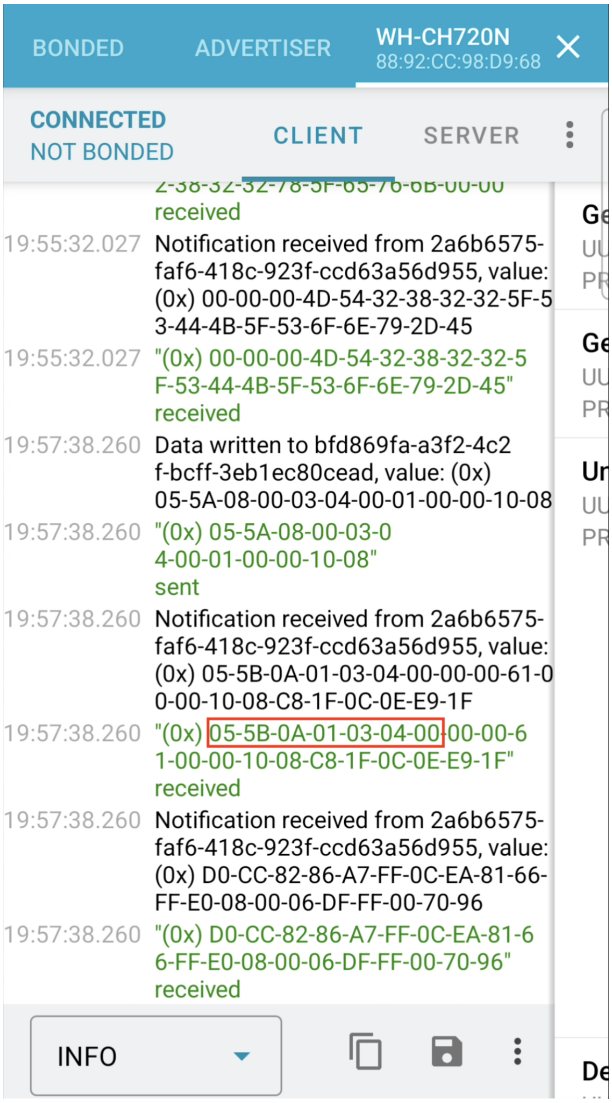

Figure 36: Android Log

Conclusion

If these steps were successful, your device is vulnerable. An attacker can read out sensitive data from the device's flash and potentially use other RACE commands. Check whether the vendor of your device offers a firmware update.

6.2 RACE Toolkit

The RACE Toolkit can be found on GitHub[8]. It is our implementation of a subset of RACE commands, some of which are described in this white paper, as well as an implementation of some of its transports.

The toolkit is largely aimed at researchers that like to do research on Airoha based devices. It also offers command to check whether a given device is vulnerable to the vulnerabilities outlined in Section 5.

Details on the setup of the tool can be found in the `Readme.md` that resides in the repository. Once installed, the `check` command is run as follows:

```
python race_toolkit.py check
```

The command will interactively guide the user through the process. It performs the following actions:

1. Scan BLE devices and inquire whether any of the identified devices belongs to the user.
2. Connect to the device and enumerate GATT services.
3. Check if one of the known the RACE GATT service UUIDs is present.
4. Test the Read Flash RACE command.
5. Try to obtain the Bluetooth Classic address via the respective RACE command.
   - Not all devices support this command. If it fails, it will ask the user for the Bluetooth Classic address.
6. Connect to the device via Bluetooth Classic.
7. Enumerate RFCOMM services.
8. Check if one of the known RACE RFCOMM services is present.
9. Connect and attempt to read flash using the RACE command via RFCOMM.

If the device's *Bluetooth Classic address* is known, it can be supplied via the `--target-address` parameter. The toolkit will try to obtain the address during the BLE phase. In case this fails, it will interactively ask for the address. If the device is not available via BLE the automatic retrieval of the Classic address will not work. If no Bluetooth Classic address is given or identified, the RFCOMM based checks will not be executed.

At the end, a summarized vulnerability status is printed.

For a vulnerable device, running the `check` command might look as follows:

```
INFO: Starting device check.
INFO: Step 1: Scanning Bluetooth Low Energy devices.
INFO: Scanning for 5 seconds...
INFO: Device initialized.
INFO: Found 11 devices:
INFO: [0]: XXXXXXXXXXXXXXXX (XXXXXXXXXXXXXXXXX)
```

```
INFO: [1]: XXXXXXXXXXXXXXXX (XXXXXXXXXXXXXXXXX)
INFO: [2]: WH-CH720N (XX:XX:XX:XX:XX:XX/P)
[...]
INFO: [10]: XXXXXXXXXXXXXXXX (XXXXXXXXXXXXXXXXX)
INFO: [X]: None of these devices is mine.
INFO: Which one of these is yours?
2
INFO: Your device is WH-CH720N. Trying to identify RACE UUIDs via GATT.
INFO: Found 13 services. Checking for RACE UUIDs
INFO: Found the Sony RACE UUID 5052494D-2DAB-0341-6972-6F6861424C45!
INFO: Initiating a proper BLE connection to WH-CH720N on XX:XX:XX:XX:XX:XX/P.
INFO: Negotiated GATT MTU to 242.
INFO: Found the Sony RACE UUID 5052494D-2DAB-0341-6972-6F6861424C45!
INFO: Trying to read flash via BLE.
Dumping Flash: ████████████████████████████100%|| 16/16 [00:01<00:00, 14.63page/s]
INFO: Flash dump completed successfully.
INFO: Trying to obtain the Bluetooth Classic address for next step.
INFO: Got Bluetooth Classic address XXXXXXXXXXXX
INFO: Step 2: Checking Bluetooth Classic connection
INFO: Device initialized.
INFO: Trying to find RACE SSP RFCOMM UUID.
INFO: Found RACE UUID 8901DFA8-5C7E-4D8F-9F0C-C2B70683F5F0 for vendor SONY
INFO: Checking Bluetooth Classic Pairing Issue by initiating an HfP connection.
INFO: Connection was successful without pairing!
INFO: Trying to connect to RFCOMM RACE interface.
INFO: Device initialized.
INFO: Channel found: 21
INFO: Connected to XXXXXXXXXXXX on channel 21
INFO: Trying to read flash via Bluetooth Classic.
Dumping Flash: ████████████████████████████100%|| 16/16 [00:00<00:00, 21.76page/s]
INFO: Flash dump completed successfully.
INFO: Vulnerability status summary:
INFO:   [VULNERABLE] CVE-2025-20700: Missing GATT authentication
INFO:   [VULNERABLE] CVE-2025-20701: Missing BR/EDR authentication
INFO:   [VULNERABLE] CVE-2025-20702_LE: RACE Protocol via BLE
INFO:   [VULNERABLE] CVE-2025-20702_BR_EDR: RACE Protocol via Bluetooth Classic
```

7 Glossary

BD_ADDR (Bluetooth Device Address): A unique 6-byte identifier assigned to a Bluetooth interface, functioning similarly to a MAC address.

Bluetooth BR/EDR (Classic): The original Bluetooth protocol stack designed for high-throughput, continuous data streaming such as audio.

BLE (Bluetooth Low Energy): A power-optimized protocol stack designed for IoT applications. It operates independently from the Classic stack.

FOTA (Firmware Over-The-Air): A custom mechanism for wirelessly distributing and installing firmware updates on Airoha-based devices.

GATT (Generic Attribute Profile): The hierarchical layer in BLE used to organize data into Services and Characteristics for application data.

HCI (Host Controller Interface): The standardized interface protocol that enables communication between the host stack and the controller.

HFP (Hands-Free Profile): A standard Bluetooth profile enabling two-way audio communication and interaction between a phone and the audio device.

L2CAP (Logical Link Control and Adaptation Protocol): The protocol layer responsible for multiplexing, segmenting, and reassembling data packets for both BLE and Classic stacks.

Link Key: A shared secret key generated during the pairing process in Bluetooth Classic, used to authenticate and encrypt subsequent connections.

LMP (Link Manager Protocol): The control protocol in Bluetooth Classic responsible for link setup and control between Bluetooth devices.

OUI (Organizationally Unique Identifier): The first 24 bits of a BD_ADDR that identify the device manufacturer.

RFCOMM: A transport protocol that emulates a serial data stream over the L2CAP layer in Bluetooth Classic.

SMP (Security Manager Protocol): The protocol used in BLE to generate and distribute encryption keys and handle pairing. Unlike LMP, it runs on the host stack.

SoC (System on Chip): An integrated circuit that consolidates the Bluetooth radio, microcontroller, and memory into a single package.

TWS (True Wireless Stereo): An audio technology where left and right earbuds operate as independent wireless units, communicating with each other and the audio source.

UUID (Universally Unique Identifier): A 128-bit value used to uniquely identify Services, Characteristics, and specific data channels within Bluetooth.

8 References

[1] AIROHA Technology Corp. *Airoha Technology*. https://www.airoha.com/.

[2] AIROHA Technology Corp. *Product Security Bulletin 2025 | Airoha Technology*. https://www.airoha.com/product-security-bulletin/202.

[3] bluekitchen. *GitHub BTstack Example HfP HF Demo*. https://github.com/bluekitchen/btstack/blob/master/example/hfp_hf_demo.c.

[4] BlueKitchen GmbH. *BlueKitchen – BlueKitchen GmbH | Qualified Dual-mode Bluetooth Stack – Available in Source Code*. https://bluekitchen-gmbh.com/.

[5] CCC. *Bluetooth Headphone Jacking: A Key to Your Phone*. https://events.ccc.de/congress/2025/hub/en/event/detail/bluetooth-headphone-jacking-a-key-to-your-phone.

[6] Frieder Steinmetz Dennis Heinze. *Headphone Jacking: A Key to Your Phone*. https://troopers.de/downloads/troopers25/TR25_Headphone_Jacking_FBNB8Y.pdf/.

[7] Frieder Steinmetz Dennis Heinze. *Part I: Bluetooth Auracast from a Security Researcher's Perspective*. https://insinuator.net/2025/01/auracast-part1/.

[8] Frieder Steinmetz Dennis Heinze. *RACE Toolkit*. https://github.com/auracast-research/race-toolkit.

[9] Frieder Steinmetz Dennis Heinze. *Security Advisory: Airoha-based Bluetooth Headphones and Earbuds*. https://insinuator.net/2025/06/airoha-bluetooth-security-vulnerabilities/.

[10] GN Audio. *Jabra Security Center*. https://www.jabra.com/supportpages/security-center.

[11] GN Audio. *Release Notes for Jabra Elite 8 Active*. https://www.jabra.com/support/release-notes/release-note-jabra-elite-8-active.

[12] Great Scott Gadgets. *Ubertooth One*. https://greatscottgadgets.com/ubertoothone/.

[13] I love audio network. *52Audio*. https://www.52audio.com/.

[14] MOOR Technology Co., Ltd. *TV HearMore Auracast Audio Transmitter*. https://www.moor-audio.com/products/tv-hearmore-bluetooth-auracast-audio-transmitter.

[15] ramikg. *Airoha firmware parser & decryptor*. https://github.com/ramikg/airoha-firmware-parser.

[16] TARLOGIC. *BSAM: Bluetooth Security Assessment Methodology*. https://www.tarlogic.com/bsam/.