Final Project Report

Reverse-Engineering Ray-Ban Metas

34371 Communication Network Security, Fall 2025

Michael Xu (s251986)
Hussein Elguindi (s252091)

December 31, 2025

Table of Contents

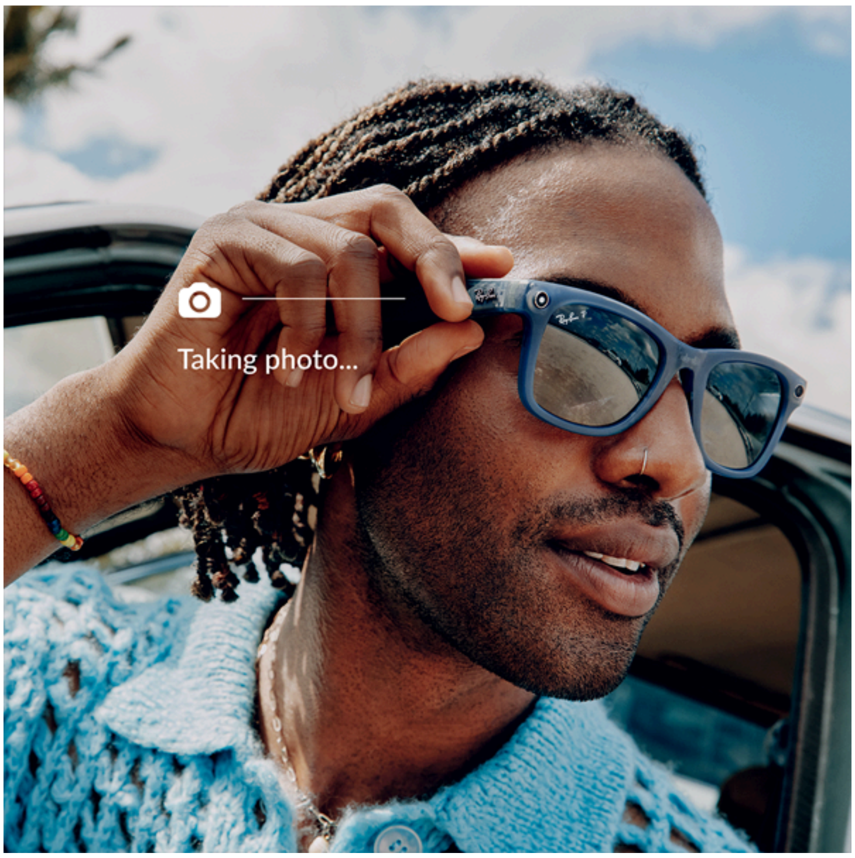
Abstract

The Ray-Ban Meta smart glasses represent a significant evolution in consumer IoT, integrating high-definition video capture, open-ear audio, and AI assistants into traditional eyewear frames. While these devices offer seamless connectivity through the proprietary Meta View companion app, their heavy reliance on wireless data transmission raises critical questions regarding user privacy and protocol security.

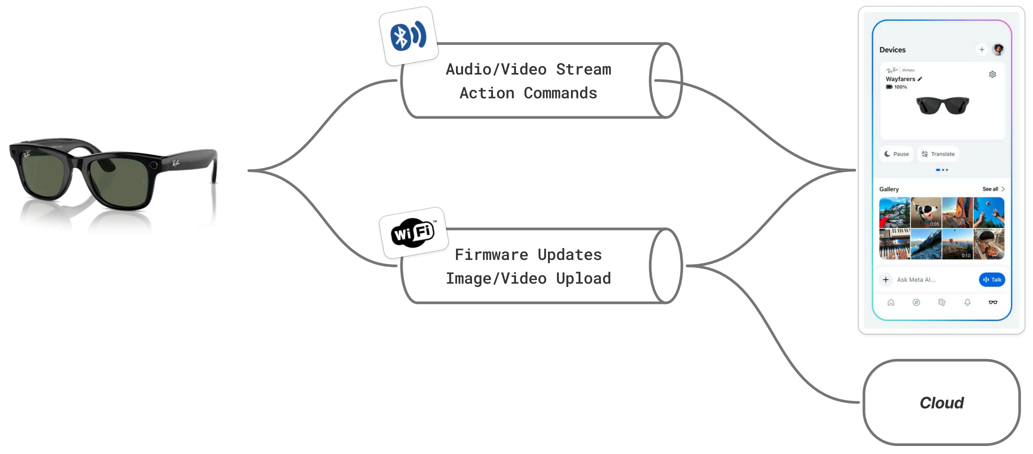
This project aims to reverse-engineer the communication stack of the Ray-Ban Meta glasses to assess their resilience against unauthorized interception and manipulation. Our research focuses on two primary security objectives:

1. **Unsanctioned Media Extraction:** Demonstrating the feasibility of intercepting and extracting audio or video payloads directly from the device without authentication from the companion application.
2. **Remote Command Injection:** Attempting to trigger device actions—such as pausing media or modifying volume levels on the paired device—without physical interaction or valid app commands.

To achieve this, we will analyze the device's Bluetooth protocol stack, with a specific focus on RTP (Real-time Transport Protocol), AVCTP (Audio/Video Control Transport Protocol), A2DP, L2CAP, and BLE. Additionally, we aim to inspect the Wi-Fi Direct mechanisms utilized for high-bandwidth media synchronization, time permitting. This report documents our methodology, packet analysis, and the security implications of our findings.



Technique Intro



Security Framework & Mechanisms

The Ray-Ban Meta smart glasses utilize a hybrid "Dual-Homing" network architecture to balance power consumption with bandwidth requirements. The security framework is divided across two primary protocols:

Bluetooth Low Energy (BLE) for Control

- The device uses BLE for the initial handshake, status updates (battery, connectivity), and command execution.
  - **Privacy**: To prevent persistent tracking, the glasses utilize **Resolvable Private Addresses (RPA)**, rotating their MAC address periodically (e.g., `73:ee:69...` to `98:ab:54...`).
  - **Encryption**: The pairing process adheres to the **LE Secure Connections** standard, utilizing **Elliptic Curve Diffie-Hellman (ECDH)** for key exchange. This protocol ensures that shared session keys are generated mathematically on both devices without being transmitted over the air, protecting against passive eavesdropping.

Wi-Fi for Data Transport

- For high-bandwidth tasks like "Live Video Streaming" and "Importing Images," the glasses initiate a **Protocol Handoff**. The glasses act as a SoftAP (Software Access Point), and the smartphone connects to this temporary Wi-Fi network to transfer media files.

Usage in IoT

This architecture represents a classic "Companion Device" model in IoT. The smart glasses act as a headless peripheral that relies on a central gateway (the smartphone running the Meta View app) for internet connectivity and heavy processing. This creates a dependency where the security of the IoT device is intrinsically linked to the integrity of the pairing process with the gateway.

Vulnerabilities

While BLE Secure Connections (Mode 1, Level 4) generally provides strong encryption against passive sniffing, the security of the connection relies heavily on the **Association Model** used during pairing. The strength of the authentication is determined by the I/O capabilities of the devices (e.g., presence of a screen or keyboard).

Case 1: The "Just Works" Downgrade (MITM)

The most critical vulnerability identified in our reverse engineering is the reliance on the **"Just Works"** pairing association model. Because the Ray-Ban Metas lack a display screen or a keypad, they cannot perform "Passkey Entry" or "Numeric Comparison" to verify the identity of the connecting device.

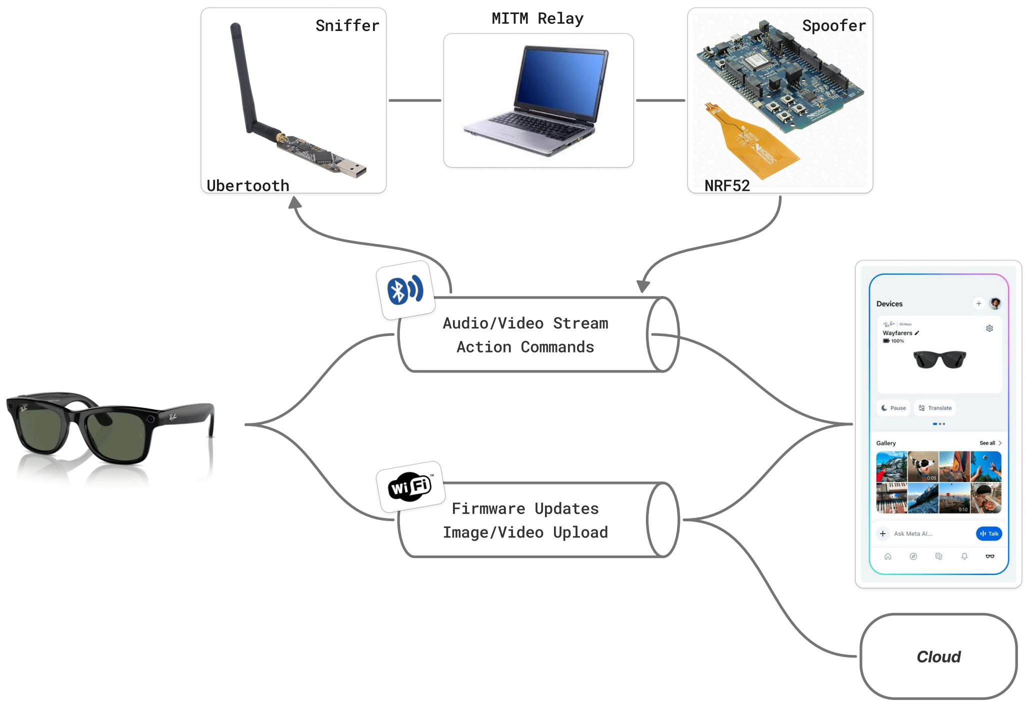
- **The Flaw**: The Bluetooth specification mandates that "No Input / No Output" devices default to "Just Works." In this mode, the cryptographic nonce exchange is unauthenticated; the devices effectively "auto-accept" the connection without user verification.
- **Impact**: This leaves the device susceptible to **Man-in-the-Middle (MITM)** attacks. An attacker can impersonate the phone to the glasses and the glasses to the phone, intercepting all control traffic.

Case 2: Wi-Fi De-authentication & Protocol Handoff

The handoff mechanism for media transfer introduces a vulnerability in the availability and integrity of the data stream.

- **The Flaw**: The Wi-Fi SoftAP created by the glasses operates on standard 802.11 protocols which are susceptible to management frame attacks.
- **Impact**: As demonstrated, the link is vulnerable to **De-authentication** attacks. An attacker can inject de-auth frames to forcibly disconnect the smartphone from the glasses during a live stream or file import, causing a Denial of Service (DoS). Furthermore, if the WPA2 credentials (exchanged over the potentially compromised BLE link) are intercepted, an attacker could join the network and access the media files directly.

Attacker's Perspective: Experiments to Reproduce Attacks



Required Tools

To validate these vulnerabilities, we designed a lab setup comprising three distinct roles:

1. **Sniffer**: An **Ubertooth One** used for passive reconnaissance to identify advertising channels and capture the initial handshake packets.
2. **MITM Relay**: A laptop acting as the central processing node for the attack logic.
3. **Spoofer**: An **nRF52 Development Kit** running the **Zephyr OS**. This hardware allows for simultaneous "Central" and "Peripheral" roles, essential for the active relay attack.

Software Stack

- **Wireshark, PacketLogger, & Ubertooth tools**: For packet analysis.
- **Aircrack-ng**: For performing the Wi-Fi de-authentication attack.
- **Jadx**: Used to decompile the Meta AI Android application to map the GATT services and characteristics.

Attack Flow

The reproduction of the MITM attack follows this execution flow:

1. **Reconnaissance**: Use the Ubertooth to sniff Advertising Indication packets (`ADV_IND`) and identify the target's MAC address and specific Manufacturer Data.
   - See our code to do this with native device Bluetooth capability
   - We also experimented with Host Controller Interface (HCI) sniffing using Apple's PacketLogger. This was useful for protocol discovery, though it was not useful for extracting data as that is not logged.

2. **Cloning**: Configure the nRF52 Spoofer to broadcast the exact same UUIDs and Device Name as the real Ray-Bans.
3. **Jamming & Disconnect**: Force a disconnection of the real glasses (using signal jamming or simply waiting for the user to cycle power).
4. **The Relay (MITM)**:
   - The Victim (iPhone) scans and connects to our nRF52 (thinking it is the glasses).
   - The nRF52 immediately initiates a connection to the Real Glasses.
   - Due to **"Just Works"** pairing, the iPhone auto-confirms the connection without requesting a code.
5. **Interception**: The nRF52 now forwards packets between the two devices, logging sensitive GATT commands (like "Start Streaming" or Wi-Fi credentials) in the process.

Defender’s Perspective: Countermeasures

To mitigate the vulnerabilities identified in our “Just Works” MITM attack and Wi-Fi handoff analysis, the following countermeasures are recommended:

Implement Out-of-Band (OOB) Pairing

Since the glasses lack a screen, Meta should utilize OOB pairing to secure the BLE link. This could involve an NFC tag embedded in the charging case or a QR code in the packaging that contains the encryption keys. This forces authentication that an MITM attacker cannot replicate without physical access.

Application-Level Encryption

Do not rely solely on Bluetooth or WPA2 transport security. Sensitive data (like authentication tokens or media) should be encrypted at the application layer before transmission. Even if the BLE link is bridged by an attacker, the payload would remain opaque.

Management Frame Protection (802.11w)

To prevent the De-authentication attacks against the Live Video Stream, the SoftAP implementation should enforce 802.11w (Protected Management Frames), which prevents unauthorized devices from sending disconnect commands.

Individual Contributions

Michael

- Researched the Bluetooth "Just Works" association model to identify the lack of authentication in IO-constrained devices.
- Studied the 802.11 management frame structure to understand how de-authentication frames could disrupt the SoftAP handoff.
- Configured the nRF52 Development Kit with Zephyr OS to act as a spoofer, replicating the glasses' UUIDs and Device Name.
- Attempted to execute the "Just Works" MITM attack simulation to demonstrate how a spoofer could intercept the connection.
- Conducted passive packet sniffing using the Ubertooth One to capture initial advertising packets and analyze MAC address rotation.

Hussein

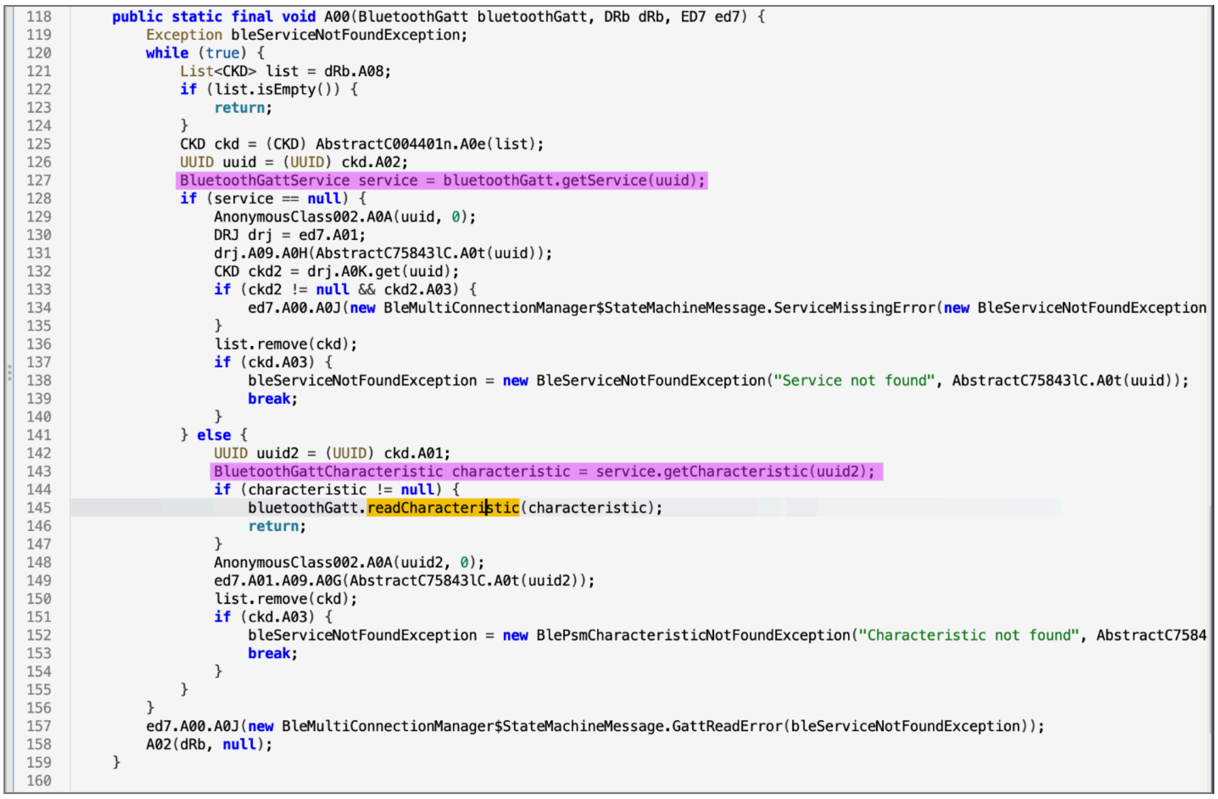
- Reverse-engineered the "Meta View" Android application using **Jadx** to inspect the security logic.
- Wrote and tested the `ble_characteristics.py` Python script using the `Bleak` library to successfully discover and read from the glasses' GATT server without the companion app.
- Analyzed the `BluetoothManager` class to understand how the app handles GATT services.
- Mapped the obfuscated GATT characteristic UUIDs to the internal "Protocol/Service Manager" (PSM) definitions (e.g., mapping specific UUIDs to Firmware and Battery services).
- Verified the existence of these hidden endpoints by cross-referencing decompiled code with live Bluetooth scans.
- Conducted passive packet sniffing using the Ubertooth One to capture initial advertising packets and analyze MAC address rotation.

Companion App Decompilation

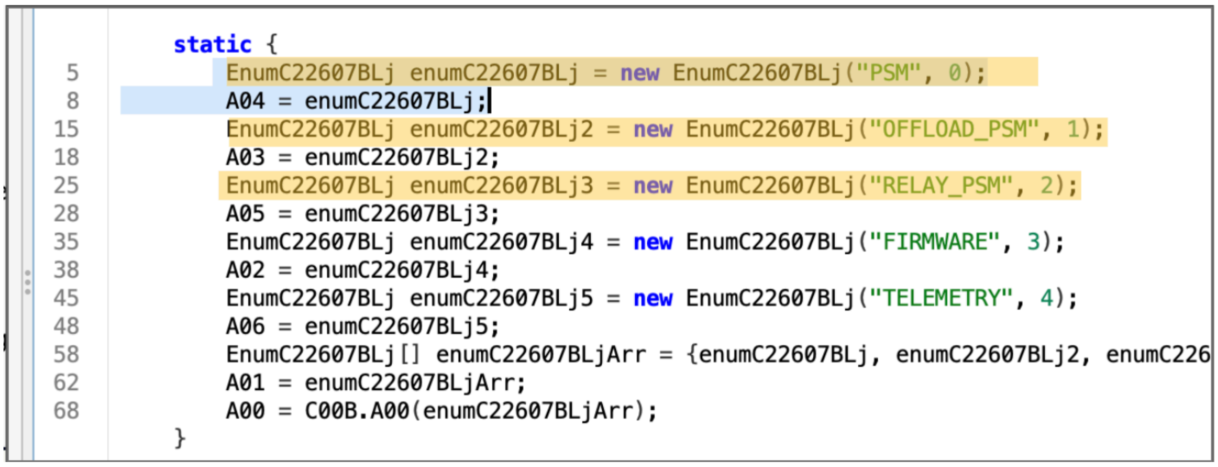
We used the Jadx Java decompiler to decompile the Meta AI companion app.

GATT Endpoints and Characteristics

We were able to pinpoint parts of the code with handle GATT and BLE characteristics. In addition, we found that the app abstracts these characteristics into what they call a Protocol/Service Manager (PSM).

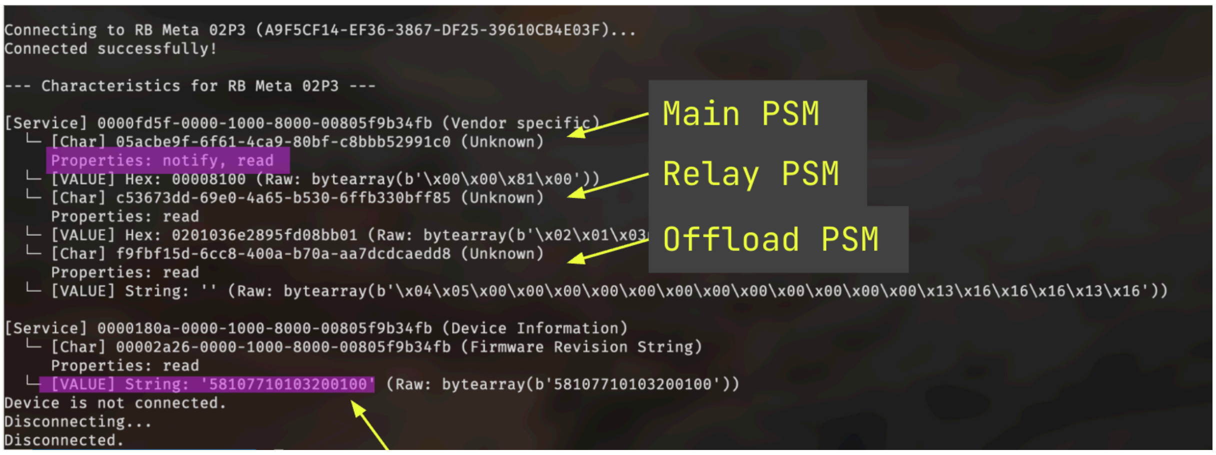


```java
    public static final void A00(BluetoothGatt bluetoothGatt, DRb dRb, ED7 ed7) {
        Exception bleServiceNotFoundException;
        while (true) {
            List<CKD> list = dRb.A08;
            if (list.isEmpty()) {
                return;
            }
            CKD ckd = (CKD) AbstractC004401n.A0e(list);
            UUID uuid = (UUID) ckd.A02;
            BluetoothGattService service = bluetoothGatt.getService(uuid);
            if (service == null) {
                AnonymousClass002.A0A(uuid, 0);
                DRJ drj = ed7.A01;
                drj.A09.A0H(AbstractC75843lC.A0t(uuid));
                CKD ckd2 = drj.A0K.get(uuid);
                if (ckd2 != null && ckd2.A03) {
                    ed7.A00.A0J(new BleMultiConnectionManager$StateMachineMessage.ServiceMissingError(new BleServiceNotFoundException
                }
                list.remove(ckd);
                if (ckd.A03) {
                    bleServiceNotFoundException = new BleServiceNotFoundException("Service not found", AbstractC75843lC.A0t(uuid));
                    break;
                }
            } else {
                UUID uuid2 = (UUID) ckd.A01;
                BluetoothGattCharacteristic characteristic = service.getCharacteristic(uuid2);
                if (characteristic != null) {
                    bluetoothGatt.readCharacteristic(characteristic);
                    return;
                }
                AnonymousClass002.A0A(uuid2, 0);
                ed7.A01.A09.A0G(AbstractC75843lC.A0t(uuid2));
                list.remove(ckd);
                if (ckd.A03) {
                    bleServiceNotFoundException = new BlePsmCharacteristicNotFoundException("Characteristic not found", AbstractC7584
                    break;
                }
            }
        }
        ed7.A00.A0J(new BleMultiConnectionManager$StateMachineMessage.GattReadError(bleServiceNotFoundException));
        A02(dRb, null);
    }
```



```java
    static {
        EnumC22607BLj enumC22607BLj = new EnumC22607BLj("PSM", 0);
        A04 = enumC22607BLj;
        EnumC22607BLj enumC22607BLj2 = new EnumC22607BLj("OFFLOAD_PSM", 1);
        A03 = enumC22607BLj2;
        EnumC22607BLj enumC22607BLj3 = new EnumC22607BLj("RELAY_PSM", 2);
        A05 = enumC22607BLj3;
        EnumC22607BLj enumC22607BLj4 = new EnumC22607BLj("FIRMWARE", 3);
        A02 = enumC22607BLj4;
        EnumC22607BLj enumC22607BLj5 = new EnumC22607BLj("TELEMETRY", 4);
        A06 = enumC22607BLj5;
        EnumC22607BLj[] enumC22607BLjArr = {enumC22607BLj, enumC22607BLj2, enumC226
        A01 = enumC22607BLjArr;
        A00 = C00B.A00(enumC22607BLjArr);
    }
```

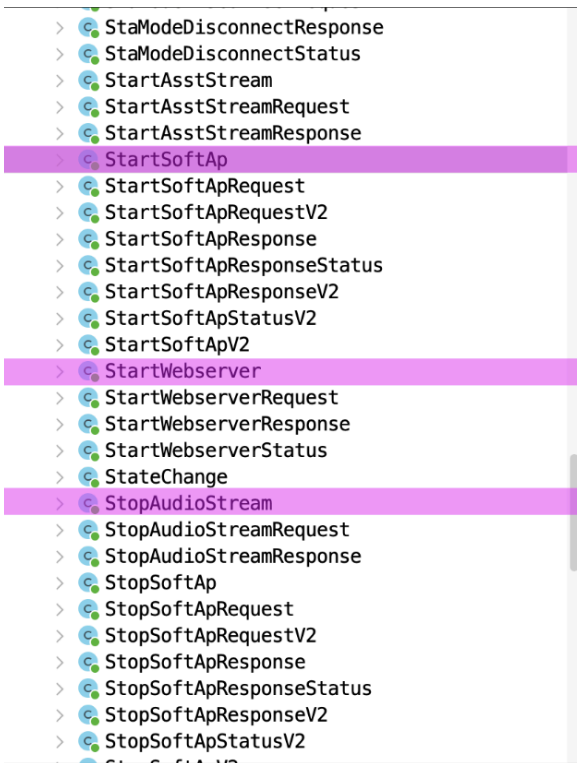
Notice there is an enum for firmware, in addition to the other PSMs. We were able to read the firmware version from this endpoint.



```
Connecting to RB Meta 02P3 (A9F5CF14-EF36-3867-DF25-39610CB4E03F)...
Connected successfully!

--- Characteristics for RB Meta 02P3 ---

[Service] 0000fd5f-0000-1000-8000-00805f9b34fb (Vendor specific)
  └ [Char] 05acbe9f-6f61-4ca9-80bf-c8bbb52991c0 (Unknown)
    Properties: notify, read
  └ [VALUE] Hex: 00008100 (Raw: bytearray(b'\x00\x00\x81\x00'))
  └ [Char] c53673dd-69e0-4a65-b530-6ffb330bff85 (Unknown)
    Properties: read
  └ [VALUE] Hex: 0201036e2895fd08bb01 (Raw: bytearray(b'\x02\x01\x03
  └ [Char] f9fbf15d-6cc8-400a-b70a-aa7dcdcaedd8 (Unknown)
    Properties: read
  └ [VALUE] String: '' (Raw: bytearray(b'\x04\x05\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x13\x16\x16\x16\x13\x16'))

[Service] 0000180a-0000-1000-8000-00805f9b34fb (Device Information)
  └ [Char] 00002a26-0000-1000-8000-00805f9b34fb (Firmware Revision String)
    Properties: read
  └ [VALUE] String: '58107710103200100' (Raw: bytearray(b'58107710103200100'))
Device is not connected.
Disconnecting...
Disconnected.
```

We used our characteristics scanner to map each exposed GATT endpoint to a PSM in our decompilation. We were able to read from each endpoint. The PSMs were encrypted, but the firmware version was unobfuscated and matched the version reported in the app.

Finally, we confirmed that the glasses started a WiFi AP and webserver to perform data transfer such as importing images from the glasses to the companion device.



```
StaModeDisconnectResponse
StaModeDisconnectStatus
StartAsstStream
StartAsstStreamRequest
StartAsstStreamResponse
StartSoftAp
StartSoftApRequest
StartSoftApRequestV2
StartSoftApResponse
StartSoftApResponseStatus
StartSoftApResponseV2
StartSoftApStatusV2
StartSoftApV2
StartWebserver
StartWebserverRequest
StartWebserverResponse
StartWebserverStatus
StateChange
StopAudioStream
StopAudioStreamRequest
StopAudioStreamResponse
StopSoftAp
StopSoftApRequest
StopSoftApRequestV2
StopSoftApResponse
StopSoftApResponseStatus
StopSoftApResponseV2
StopSoftApStatusV2
```

Code

`ble_characteristics.py`

Archived as a GitHub Gist

```python
"""
Tool to get BLE device characteristics.
"""

import asyncio
from bleak import BleakScanner, BleakClient

class DeviceBle:
    def __init__(self):
        self.client = None
        self.device = None

    async def select_device(self):
        print("Scanning for devices (5 seconds)...")
        # 1. Discover all devices
        # We sort by RSSI (Signal Strength) so the closest device is likely at the top
        devices = await BleakScanner.discover(timeout=5.0)
        # devices.sort(key=lambda d: d.rssi, reverse=True)

        if not devices:
            print("No Bluetooth devices found.")
            return False

        # 2. Print the list
        print("\n--- Available Devices ---")
        # Filter out devices to create a selection list
        selectable_devices = []
        for d in devices:
            # You can remove the 'if d.name' check if you want to see unnamed devices (MAC only)
            if d.name:
                selectable_devices.append(d)

        if not selectable_devices:
            print("No named devices found.")
            return False

        for i, dev in enumerate(selectable_devices):
            print(f"[{i}] {dev.name} ({dev.address})")

        # 3. Get User Input
        while True:
            try:
                selection = input("\nEnter the number of the device to connect to: ")
                index = int(selection)
                if 0 <= index < len(selectable_devices):
                    self.device = selectable_devices[index]
                    return True
                else:
                    print("Invalid number. Please try again.")
            except ValueError:
                print("Please enter a valid number.")

    async def connect(self):
        if self.device is None:
            print("No device selected.")
            return

        print(f"\nConnecting to {self.device.name} ({self.device.address})...")
        try:
            self.client = BleakClient(self.device.address)
            await self.client.connect()
            print("Connected successfully!")
        except Exception as e:
            print(f"Failed to connect: {e}")
            self.client = None
    async def disconnect(self):
        if self.client and self.client.is_connected:
            print("Disconnecting...")
            await self.client.disconnect()
            print("Disconnected.")

    def decode_bytes(self, byte_data):
        """Helper to try and make bytes readable"""
        # 1. Try to decode as UTF-8 String (e.g. Model Name)
        try:
            return f"String: '{byte_data.decode('utf-8')}'"
        except Exception:
            pass

        # 2. If not string, return Integer if small
        if len(byte_data) == 1:
            return f"Int: {int.from_bytes(byte_data, byteorder='little')}"

        # 3. Fallback to Hex
        return f"Hex: {byte_data.hex()}"

    async def print_all_characteristics(self):
        if self.client and self.client.is_connected:
            print(f"\n--- Characteristics for {self.device.name} ---")
            for service in self.client.services:
                print(f"\n[Service] {service.uuid} ({service.description})")
                for char in service.characteristics:
                    print(f"  └─ [Char] {char.uuid} ({char.description})")
                    print(f"     Properties: {', '.join(char.properties)}")
                    # CHECK: Can we read this?
                    if "read" in char.properties:
                        try:
                            # READ THE DATA
                            value_bytes = await self.client.read_gatt_char(char.uuid)

                            # FORMAT THE DATA
                            decoded_val = self.decode_bytes(value_bytes)

                            print(f"  └─ [VALUE] {decoded_val} (Raw: {value_bytes})")
                        except Exception as e:
                            print(f"  └─ [ERROR] Could not read: {e}")
                    else:
                        print(f"  └─ [SKIP] Not readable")
            print("Device is not connected.")

async def main():
    ble_handler = DeviceBle()

    # Step 1: User selects device from list
    device_selected = await ble_handler.select_device()

    if device_selected:
        # Step 2: Connect
        await ble_handler.connect()

        # Step 3: Print Characteristics
        await ble_handler.print_all_characteristics()

        # Step 4: Clean up
        await ble_handler.disconnect()

if __name__ == "__main__":
    asyncio.run(main())
```

Sources

- A new way to debug iOS Bluetooth® applications (PacketLogger)
- Audio/Video Control Transport Protocol (AVTCP) Specification
- Ubertooth Wireshark documentation
- Meta AI companion app APK source
- GitHub - NullPxl/banrays: Glasses to detect smart-glasses that have cameras. Ray-BANNED
- Assigned Numbers
- THREAT ASSESSMENT ALERT: THE DANGERS OF SMART GLASSES
- Meta FAQs | Ray-Ban® US
- Bluetooth: Central and Peripheral HRS
- Samples and Demos — Zephyr Project Documentation
- deauthentication [Aircrack-ng]
- Bluetooth® LE secure connections – numeric comparison
- Download Meta AI - Vibes & AI Glasses APKs for Android - APKMirror
- GitHub - skylot/jadx: Dex to Java decompiler
- Ubertooth
- AWUS036NHA (EOL) – ALFA Network Inc.