Revisiting Bluetooth Vulnerabilities in Automotive Infotainment Systems: A Remote Code Execution Case Study

Philippe Trébuchet
Hardware and Software Architectures Lab
National Cybersecurity Agency of France (ANSSI)
Paris, France
philippe.trebuchet@ssi.gouv.fr

Guillaume Bouffard
Hardware and Software Architectures Lab
National Cybersecurity Agency of France (ANSSI)
Paris, France
guillaume.bouffard@ssi.gouv.fr

***Abstract*—Modern vehicles integrate multiple wireless interfaces, such as Bluetooth, which significantly expand their attack surface. In this work, we analyze a proprietary Bluetooth stack embedded in a widely deployed automotive infotainment system. We show that this implementation contains a vulnerability similar to `CVE-CVE-2018-20378`, despite vendor claims that the studied version is unaffected.**

We demonstrate the exploitability of this vulnerability by developing specially crafted exploitation primitives, achieving remote code execution without user interaction, despite the presence of mitigations such as Address Space Layout Randomization (ASLR) and Data Execution Prevention (DEP).

Our results highlight a gap between publicly documented vulnerabilities and their practical exploitability in real-world embedded systems.

***Index Terms*—Automotive security, Bluetooth, embedded systems security, vulnerability analysis, remote code execution**

1. Introduction

Modern vehicles, commonly referred to as *connected cars*, have evolved into true computers on wheels. Through their infotainment systems, they integrate numerous wireless communication technologies such as Bluetooth, Wi-Fi, and cellular networks. Though improving the comfort of the passenger, this increasing interconnectivity also introduces new cybersecurity risks.

The Bluetooth stack, in particular, has become a central component of automotive infotainment and telematics systems. Yet, its implementations in vehicles may expose vulnerabilities that can be exploited remotely [1]–[4].

The security of connected vehicles [5]–[8] has thus become a major concern for manufacturers and regulators [9]. A compromise of these systems can have severe consequences, ranging from privacy violations (e.g., stealth geolocation, in-cabin eavesdropping) to physical safety risks for vehicle occupants and bystanders.

The intrinsic *always on* characteristic of Bluetooth makes it a particularly sensitive protocol from a security standpoint, as each potential connection represents a possible entry point.

In this paper, we analyze the security of a Bluetooth stack embedded in the infotainment system of a modern vehicle. The same implementation is deployed across multiple models from different manufacturers [10]. Our analysis revealed a vulnerability that allows a remote attacker to take full control of the infotainment system *without any user interaction*, i.e. a 0-click RCE.

1.1. Contributions

This paper makes the following contributions:

1) We present a full black-box security analysis of a Bluetooth stack deployed in a production automotive infotainment system representative of early 2020s vehicles.
2) We demonstrate that this implementation contains a programming flaw similar to `CVE-CVE-2018-20378`, despite vendor claims that the embedded version falls outside the affected range. We revisit the root cause and show that the vulnerability remains exploitable in this context.
3) We develop new exploitation primitives that enable a controlled buffer overflow and arbitrary memory writes.
4) We construct a complete exploitation chain that bypasses Address Space Layout Randomization (ASLR) and Data Execution Prevention (DEP), leading to remote code execution on the infotainment system.
5) We evaluate the security impact of this compromise, showing that it enables Controller Area Network (CAN) message injection and control over in-vehicle communications. We conducted a responsible disclosure process with the affected manufacturer, who confirmed and patched the issue across multiple vehicle models.

1.2. Key Difference with Prior Work

The original `CVE-2018-20378` exploitation scenario relies on very specific assumptions about memory layout, the availability of many concurrent clients, and the absence of certain mitigations. These assumptions do not hold on our target system, making the described exploitation impossible; this is likely why the examined version is listed as unaffected.

We show that the root cause remains and is exploitable by developing specialy crafted exploitation primitives,

ultimately achieving remote code execution despite the presence of standard mitigations.

This paper is organized as follows: Section 2 provides an overview of automotive cybersecurity with a focus on our target platform and attacker model. The analysis, vulnerability discovery, and exploitation are detailed in Section 3. The impact of our findings is discussed in Section 4 before concluding and outlining future directions in Section 5.

2. Security of Connected Vehicles

2.1. Target Description

In this work, we focus on a premium vehicle model representative of 2020s designs. While not the latest model sold, this vehicle is widely deployed/sold and actively maintained by the manufacturer, making it a highly relevant and realistic target for evaluating the current security posture of connected vehicles.

Among the vehicle's Electronic Control Units (ECUs), the infotainment unit exposes the largest number of external interfaces. This ECU is physically connected to multiple in-vehicle communication buses and maintains a direct link with the Transmission Control Unit (TCU). Given its central role in vehicle operation, we focused our study on this unit.

On our target vehicle, the infotainment ECU runs Android 4.4 with a Linux kernel 3.0.35. At the start of our study, this class of ECU equipped many new vehicles from the targeted manufacturer.

The hardware platform integrates an i.MX 6 DualLite, 2 GB of RAM and a 32 GB eMMC storing both the system and navigation data.

Firmware analysis also revealed a relatively mature security configuration on the target platform:

- **High Assurance Boot (HAB)**. This i.MX 6 feature enforces execution only of signed code on the platform, preventing unauthorized firmware modifications.
- **ASLR**. Memory addresses used by process components are non-predictable.
- **DEP**. Several protections are enabled: non-executable stacks, `W^X`, and microarchitectural analogues such as Supervisor Mode Execution Protection (SMEP) and Supervisor Mode Access Prevention (SMAP).
- **Kernel `DEBUG` symbols** have been stripped and the kernel `config` file does not appear anywhere in the disk image.

Of course, in 2020, this inventory lacks many defenses introduced in later Android releases. However, given the hardware manufacturing date and the Android version present, we think that the available security features were enabled. To compare, in [11] platforms that are four years newer are mentioned, yet ASLR was found to be disabled.

Regarding the infotainment ECU's attack surface, it offers Bluetooth and Wi-Fi connectivity. The kernel version suggests a number of unpatched Common Vulnerabilities and Exposures (CVE) are likely present, particularly in drivers for external interfaces (Bluetooth, USB and Wi-Fi), and the age of its components implies a non-negligible number of known vulnerabilities.

The Bluetooth stack is externalized from the kernel, and the `bluego` app controls an external Bluetooth modem connected to the application System on a Chip (SoC) via an Universal Asynchronous Receiver Transmitter (UART) carrying the traffic to and from the external interface. No Bluetooth traffic is processed by the kernel. Examination of `bluego` shows that it uses the `Blue SDK` protocol stack from OpenSynergy.

2.2. Threat Model

Vehicles typically provide a diagnostic connector that grants access to the vehicle's onboard network of ECUs: the On-Board Diagnostics (OBD) port. Its primary purpose is to allow authorized personnel with appropriate software to perform maintenance operations. Vehicles often also include other connectors; for instance, a USB port linked to the infotainment unit. An attacker with physical access to the interior of the vehicle can therefore interact with virtually any component, i.e., can *de facto* perform any attack they choose, including replacing or tampering physically with one or more vehicle components.

Physical compromise of a vehicle has devastating consequences, and few practical defenses exist against an attacker located inside the vehicle. Perimeter security (alarms, guarded parking) can make such attacks difficult, and physical tampering typically leaves traces that enable later detection (operational locking systems, forensic evidence).

The adversary model considered in this study assumes an attacker who can interact with the vehicle remotely, without performing any physical manipulation of the vehicle.

Under this adversary model, the vehicle's residual attack surface is limited to its external interfaces. For our target vehicle, examples of exposed components include tire pressure sensors, front and rear cameras, proximity sensors, a GSM modem, a Wi-Fi interface, and a Bluetooth interface.

The Bluetooth protocol is presented in Appendix A.

3. MTU and Overflows

3.1. CVE-2018-20378

`CVE-2018-20378`, also known as *Hell2CAP* [11], affects `Blue SDK` versions 3.2 through 6.0. It allows unauthenticated remote attackers to execute arbitrary code by exploiting a buffer overflow triggered by incorrect handling of the Logical Link Control and Adaptation Protocol (L2CAP) channel `MTU` configuration.

3.1.1. Misconfiguration of the L2CAP `MTU`. L2CAP manages signaling and channel configuration for Bluetooth communications. When a configuration request carrying an `MTU` value arrives, `Blue SDK` first assigns the requested byte value to the channel structure *and only*

then checks whether the value is valid. This value governs the size of response packets for encapsulated upper-layer protocols.

If the value is below the Bluetooth-specified minimum of 48 bytes [12], the channel is marked invalid but not closed; the specification does not mandate a specific behavior in this case. If the client later sends valid requests, the channel remains usable despite the invalid `MTU`. Figure 1 shows a scenario where an attacker configures the L2CAP `MTU` to 20 bytes.

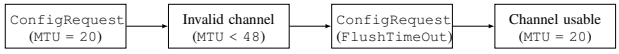

Figure 1. Compromising the L2CAP `MTU` during channel configuration. Adapted from [11].

3.1.2. Out of Bounds Writes. The Service Discovery Protocol (SDP) layer uses the `MTU` provided by L2CAP to allocate fields for its responses and to decide on packet fragmentation. According to `CVE-2018-20378`, the size of an SDP response fragment is computed as `MTU - 9` (line 4 in Listing 1), where 9 is the size of the SDP response header. If the `MTU` is less than 9 bytes, an integer underflow occurs, corrupting the maximum response size and enabling an overflow of the response buffer.

```
1 // In core/stack/sdp/sdpserv.c
2 void SdpServHandleServiceSearchAttribReq (/* ... */ ) {
3   MTU = L2CAP_GetTxMtu(_sdpInfo->CID);
4   avalaibleSizeForFragment = (MTU - 9) & 0xFFFF;
5   // ...
6   // Construct the SDP response
7   SdpStoreAttribData(_sdpInfo, _txPkt,
8                      _txPkt->buffetPtr,
9                     avalaibleSizeForFragment);
10 /* ... */ }
```

Listing 1: Source snippet from `CVE-2018-20378` highlighting the integer underflow (in red).

3.1.3. Analysis of `CVE-2018-20378`. Exploitation requires N concurrent SDP clients. The final client triggers the buffer overflow and overwrites a Function pointer (PFN) lying just after the buffer overflown. Maintaining many simultaneous clients demands careful management on the attacker side to keep connections alive, and sufficient server-side capacity to sustain a high number of parallel connections. The exploitation suggested also rely on the fact that ASLR is disabled.

Although the original `CVE-2018-20378` advisory lists versions 3.2 to 6.0 as vulnerable, the Bluetooth library embedded in our target vehicle is a **version 2.x** and was therefore assumed to be unaffected. However, as we demonstrate in the next section, this assumption is incorrect: the same vulnerability is present, but it requires a completely different exploitation approach. In the previous section, we summarized publicly available details on `CVE-2018-20378`. We now assess whether our target is susceptible to this CVE and discuss the potential consequences if it were exploitable.

Figure 2 provides an overview of our workflow. All experiments were conducted in a black-box setting with all security features enabled.

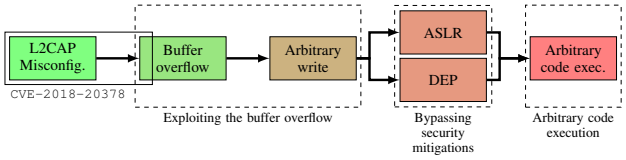

Figure 2. Overview of our workflow. Dashed boxes highlight our contributions, with labels describing each stage.

To test for `CVE-2018-20378`, we used `Scapy` to craft L2CAP packets with improper configuration values. Off-the-shelf tooling that relies on BlueZ or the Linux kernel was unsuitable here, as those stacks tend to enforce Bluetooth specification parameters [12].

To avoid interference from the Operating System (OS) Bluetooth stack, we interfaced directly at the Host Controller Interface (HCI) layer and implemented all layers up to SDP. We extended `Scapy` to support packet formats not yet available upstream.

With this setup, we confirmed that it is possible to negotiate an `MTU` value below 48, as described in Section 3.1.1, corroborating the presence of the vulnerability.

We decompose the attack into three phases, corresponding to the dashed regions in Figure 2:

- Exploiting the buffer overflow (Section 3.3)
- Bypassing security mitigations (Section 3.4)
- Arbitrary code execution (Section 3.5)

Before exploiting the buffer overflow induced by the `MTU`-related integer underflow in L2CAP, we detail relevant aspects of the Bluetooth stack implementation on the target.

3.2. Preliminaries: Analyzing the Bluetooth Stack Implementation

An incoming Bluetooth packet is first received by the external modem, then forwarded over a dedicated UART to the i.MX 6 processor. A dedicated thread in the `bluego` application receives the packet and places it into an event queue. Other worker threads process queued events and generate responses.
The Bluetooth stack and associated processing are implemented inside `libbluego.so`, which embeds `Blue SDK`.

3.2.1. SDP Server Implementation. Our analysis of `libbluego.so` shows that a static array of seven entries is allocated for each Bluetooth upper-layer protocol above HCI, in a memory region named `bt`. For each index, the array holds a structure describing per-protocol state. For SDP, this array is named `sdpServer.infos` (Listing 2, line 3) and contains entries of type `sdpServInfo`, one per connected SDP client, consistent with `CVE-2018-20378`.

The `sdpServInfo` structure forms a doubly linked list across entries of `infos`. The pointer `ptr_pkt_data` (Listing 2, line 8) references the `pkt_data` buffer at the end of the structure (Listing 2, line 14), which holds only the SDP response payload; the associated header is stored in `pkt_header`.

```
1 bt { // ..
2   sdpServer = {
3     infos[7] = {
4       // ...
5       struct sdpServInfo[0] {
6 /* 0x0000 */ struct sdpServInfo * next;
7 /* 0x0004 */ struct sdpServInfo * prev;
8 /* 0x0008 */ uint8_t * ptr_pkt_data;
9 // ...
10 /* 0x0060 */ struct sdpAttribInfo *sdpAttrib;
11 /* 0x0064 */ struct sdpRecordInfo *sdpRecords;
12 // ...
13 /* 0x0088 */ uint8_t pkt_header [5];
14 /* 0x008D */ uint8_t pkt_data [507];
15   }; // size = 648 (0x288) bytes
16       // ... Instances for sdpServInfo 1..6
17     }
18 /* 0x11D6 */ uint32_t semaphore;
19   } sdpClient {
20 /* 0x11DA */ void (*protocol_callback) (uint16_t, void *);
21       // ...
22 } };
```

Listing 2: In-memory layout of the SDP server

At Listing 2 line 20, we observe a PFN named `protocol_callback`. This function pointer is a callback used when the infotainment unit acts as an SDP client to query capabilities of a peer device. At Listing 2 line 18, a 4-byte semaphore indicates, when nonzero, that an SDP response is being constructed, thereby blocking concurrent requests to the SDP server.

This semaphore prevents the exploitation pattern described in `CVE-2018-20378` on our target. Any overflow targeting the PFN would necessarily overwrite the semaphore with uncontrolled values, making it extremely difficult to restore it to zero; the server would then stall, effectively silencing SDP and breaking the exploitation chain.

3.2.2. Triggering and Analyzing the Integer Underflow. Section 3.1 presented the integer underflow as described in `CVE-2018-20378`, where no explicit bound was enforced on the overflow size. Listing 3 shows the relevant portion of the SDP response construction on our target.

```
1 void SdpServHandleServiceSearchAttribReq ( /* ... */ )
  ↪  {
2   // ...
3   max_pkt_size = MIN(L2CAP_GetTxMtu(current_channel),
    ↪  512);
4   max_pkt_size = (max_pkt_size - 9) & 0xFFFF;
5   max_pkt_size = MIN(rx_pkt->MaximumAttributeByteCount,
    ↪  max_pkt_size);
6   // ...
7   response_size = MIN(max_pkt_size,
    ↪  total_response_len);
8 /* ... */ }
```

Listing 3: Pseudo-code illustrating the integer underflow (line 4 in red).

In this implementation (Listing 3), the maximum SDP response fragment size is determined by:

- the minimum of the channel `MTU` and 512 bytes (line 3);
- the `MTU` minus 9 bytes (line 4);
- the `Maximum Attribute Byte Count` announced by the client (line 5);
- and the total size of the SDP response (line 7).

3.3. Part 1: Exploiting the Buffer Overflow

Using `Scapy`, we connected to the target and configured the L2CAP channel `MTU` to 8. This configuration triggers an integer underflow that sets the maximum response size to the minimum of 65 535 bytes (i.e. 8 minus 9 seen as unsigned short) and the `Maximum Attribute Byte Count` field of the SDP request.

With an `MTU` of 8, regardless of the `Maximum Attribute Byte Count` value, the target builds an SDP response but does not transmit it. An internal and postponed consistency check on the SDP header size prevents the packet from being actually sent on the wire.

3.3.1. Triggering the Buffer Overflow. In the target implementation, the response buffer is statically allocated to 512 bytes: 5 bytes for the header (`pkt_header`) and 507 bytes for the data region (`pkt_data`), as shown in Listing 2. With an `MTU` of 8, which makes the maximal response, by the underflow mentionned above, as large as 65535 bytes, and with a `Maximum Attribute Byte Count` larger than 507, a buffer overflow occurs and its size is controlled by the attacker.

On our target, sending a `ServiceSearchAttributeRequest` for the L2CAP service attributes produces a response of 729 bytes. Given the 507-byte buffer, an overflow of 222 bytes ($729 - 507$) can thus be done.

3.3.2. Overwriting a Function Pointer. Referring again to Listing 2, we shaped the overflow to reach the `ptr_pkt_data` field (line 14) of `sdpServInfo[1]`, thereby corrupting the response-data pointer for a second client instance.

Using the methodology described in `CVE-2018-20378`, we triggered an overflow in the response buffer allows actual modification of this pointer. As in `CVE-2018-20378`, we used the `Continuation State` element of an SDP response, to actually write controlled bytes out of bounds. Indeed, this element is written in last position of the response buffer, and its value can be controlled by the attacker as depicted in the next lines. When the `continuation_state_info` field is non-zero (`count` > 0), it signals that additional data remain to be sent; otherwise, `count` is zero.

The target implementation supports only a single-byte `continuation_state_info`, acting as a monotonic counter from 1 to 255, which makes its value predictable.

Requiring a response larger than the `Maximum Attribute Byte Count` causes `continuation_state_info` to be written as the last byte in (or just past) the response buffer. By choosing the `Maximum Attribute Byte Count` value carefully, we can control the offset where this byte is written. This method works for writing any value except zero. Indeed, `continuation_state_info` does not take the value zero. A response packet endind with a zero means that no further information is to be expected, and this zero in the `count` value. Hence, to write a zero byte we need a different strategy: we align the `count` field of the final response packet with the targeted memory byte, as this field becomes zero when transmission completes.

Requiring a multi-packet response, and adjusting both the size of partial response packets and the total response length, while knowing the target value location, we can make the `Continuation State` fields to align the last `count` with the desired overwritten position.

Remark here that, due to the layout of the response buffer, the `Continuation State` field is always preceded by at least 16 bytes. Thus, modifying a single byte unavoidably alters these 16 preceding bytes in an uncontrolled way.

3.3.3. Building an Arbitrary Write Primitive. Once the data-pointer field `ptr_pkt_data` can be modified, arbitrary memory writes become possible.

To achieve this, we use two simultaneous Bluetooth clients, *Client 1* and *Client 2*, issuing synchronized requests.

- Client 1 triggers the buffer overflow and overwrites `ptr_pkt_data` of Client 2. Client 2 then crafts carefully chosen request packets to write attacker-controlled data to arbitrary addresses.
- Client 2 performs its writes through the `Continuation State` field, as described earlier. Unlike Client 1, it does not trigger a buffer overflow; it writes within its legitimate buffer bounds.

These two clients will be referred to as *Client 1* and *Client 2* throughout the remainder of the paper.

3.4. Part 2: Bypassing Security Mechanisms

The target deploys standard defense-in-depth mechanisms for this class of product: ASLR and DEP. DEP is enforced via `mprotect` in `libc`. The most direct way to disable DEP on a memory region is to invoke `mprotect` ourselves and relax page permissions; however, ASLR must first be defeated to find actual location of suitable gadgets and code pointers for ROP/JOP, ASLR makes these addresses otherwise unpredictable.

ASLR is enabled on our target. Base addresses for code segments, shared libraries, and each process stack are randomized. The platform is 32-bit ARM running Linux 3.0.35. We briefly review ASLR behavior for this kernel version in Appendix B.

Rapidly inferring the random base of `libbluego.so` is a prerequisite for a practical attack. The target CPU operates in little-endian mode.[1]

We first determine the high nibble of the second byte of the ASLR offset (bits 12–15 of the target address). Once recovered, we will leverage an information leak to retrieve the remaining bits.

Two observations enable recovery of this nibble:

- The lowest 12 bits of pointers are unaffected by ASLR.
- In memory, next to the `sdpServer.infos` array (line 3 in Listing 2), there exist several PFNs (function pointers) invoked at different Bluetooth protocol stages. We select one that can be dereferenced on demand and use it as an oracle, denoted `PTR_ORACLE`.

`PTR_ORACLE` must be chosen such that dereferencing him after an alteration of its value trigger a noticeable change of behavior from triggering him unaltered.

To recover the high nibble of the second offset byte, we just brute-force the 16 possible values following the two-client method of Section 3.3, Client 1 overflows to rewrite Client 2's `ptr_pkt_data` so that it points to the address holding `PTR_ORACLE` and next rewrite the targeted nibble and make `PTR_ORACLE` dereferenced.

Static analysis shows that the high nibble of Client 2's `ptr_pkt_data` second byte is `0xE`, whereas `PTR_ORACLE`'s is `0xF`. A carry in that nibble arises if the added offset makes `PTR_ORACLE` cross the nibble boundary while Client 2's `ptr_pkt_data` does not. This happens exactly when the high nibble of the ASLR offset's second byte equals 1 in that case no noticeable behavior change will occur regardless of the value written to the nibble, this lack of change will characterize this case.

Information leak. Obtaining a read primitive is challenging because responses return SDP-constructed data, not arbitrary memory. We therefore aim to have the returned data differ from what was originally written at the destination.

Knowing the high nibble of the second offset byte, Client 1 again uses the overflow to rewrite the two low bytes of Client 2's `ptr_pkt_data`, this time to make it point *into the middle* of Client 1's `pkt_data`. Since Client 1's `pkt_data` lies immediately before Client 2's `sdpServInfo`, and that the `ptr_pkt_data` of Client 2 is expected point to a 507 wide array location, a big, but legitimate request for Client 2 can the overflow Client 1 `pkt_data` and overwrite the early fields of Client 2's structure. This means that we can manage for the last written byte in the response buffer, the `Continuation State`, to overwrite the least significant byte of Client 2's `ptr_pkt_data`, this overwriting occurs just before transmission. The resulting response then spans part of Client 1's `sdpServInfo` fields. As seen in Listing 2, the pointers `sdpAttribInfo` (line 10) and `sdpRecordInfo` (line 11) follow `pkt_ptr_data`; by controlling `ptr_pkt_data`, we can capture their values.

This leakage reveals the full random offset through the leaked pointer addresses. It is not an arbitrary read primitive, since it only discloses data adjacent to Client 1's `pkt_data`, but it suffices to recover the ASLR base for our exploitation chain.

3.5. Part 3: Arbitrary Code Execution

In Section 3.3, we showed how to write to arbitrary locations within the process memory of `BlueGoe Service`. In Section 3.4, we recovered the randomized base address of `libbluego.so`. Thus we are able now forge a valid PFN (function pointer) address in order to seize control of execution.

To reach our goal we proceed in two steps. First, we identify a PFN that is both reachable and triggerable by a **remote, unauthenticated attacker with no**

1. In little-endian representation, multi-byte values are stored from least significant byte to most significant byte.

user interaction. Second, we redirect control flow to an attacker-controlled, executable region; requiring us to make memory executable.

3.5.1. Hunting for a Reachable Function Pointer. When a L2CAP connection is initiated, it is bound to a protocol identified by the request's Protocol Service Multiplexing (PSM). In the target implementation, protocol management uses the `Protocol` structure shown in Listing 4; PFNs are highlighted in red.

```
1 struct Protocol {
2   void (*proto_callback)(uint16_t, void *);
3   uint16_t psm; // Protocol type (1: SDP, 2: RFCOMM,
    ↪  ...)
4   uint16_t localMTU;   // MTU toward the peer
5   uint16_t remoteMTU; // MTU from the peer
6   // ...
7   byte (*agent_allocator)(void);
8 /* ... */ };
```

Listing 4: Protocol management implementation

Among the two callbacks defined in this structure, the second, `agent_allocator` (Listing 4, line 7), is the allocator used to create an object that manages channel lifecycle as a dedicated agent.

`agent_allocator` is invoked as soon as a connection request arrives (secure or not), and before any processing occurred, corrupting its PFN is attractive: it yields a control-flow pivot **unauthenticated and without user interaction on the target vehicle**. We therefore focus on overwriting this PFN to execute an attacker payload.

As there are many distinct protocols are handled by target device, there is as many distinct `agent_allocator` that can be corrupted.

3.5.2. Achieving arbitrary code execution. To bypass DEP, we invoke `mprotect`. `mprotect` is a member of the `libc.so` that is mapped in the process address space. Though `mprotect` is not directly referenced by any functions inside `libbluego.so` (and thus is not resolved in its symbol table), `libbluego.so` calls other `libc` routines. Consequently, `libc` is mapped and fully accessible in the process address space. Note that `libc`, as being mapped in the shared library segment, is randomized independently from `libbluego`.

In systems using the ELF binary format, dynamic binaries are those using runtime symbol resolution. Dynamic symbol resolution is handled at runtime through two tables, the Global Offset Table (GOT) and the Procedure Linkage Table (PLT), which are managed by the dynamic linker to support lazy binding or read only relocations. The BlueGo Service is such a dynamic binary. The two tables, GOT and PLT are located at a fixed offset from the code segment, i.e., as we have defeated ASLR, is at a known address.

Our plan is as follows. First, we recover the in-memory base of `libc` by reading the GOT entry of a function of the `libc` *already resolved*. Using available gadgets within `libbluego.so`, we build a short JOP-chain that reads the content of an arbitrary address, for our present purpose from the GOT, but thereby yielding a general read primitive. From the leaked function address we compute the `libc` base and then the absolute address of `mprotect`.

Second, using another short JOP chain present in `libbluego.so`, we call `mprotect` to relax permissions on a previously non-executable memory region, thus defeating DEP and enabling execution of a DATA-located shellcode.

3.5.3. End-to-End Timing. The entire attack, including the writes to stage both gadget chains and the shellcode, **completes in under 300 seconds**.

4. Security Impact

Once code execution on the infotainment system was achieved via the injected *shellcode*, we initiated a responsible disclosure process with the vehicle manufacturer. The vulnerable `Blue SDK` library we exploited is widely deployed across multiple vehicle models and manufacturers. This process enabled the vendor to coordinate internal and supplier-side investigations and deploy appropriate mitigations.

The work described in this paper was conducted between September 2023 and April 2024. The affected manufacturer was notified in September 2024. This publication was authorized with the manufacturer's agreement, under the condition that no information revealing the brand or model of the vehicle would be disclosed.

Furthermore, the manufacturer confirmed that although the presented attack enables privileged code execution on the infotainment unit and permits the transmission of CAN commands from the injected shellcode, such commands are subject to in-vehicle filtering, preventing any issuance of safety-critical or drive-control messages.

5. Conclusion

We presented a security analysis of a premium vehicle representative of early 2020s designs and still widely deployed. The target platform had all expected defense-in-depth mechanisms enabled and was fully updated with the manufacturer's available patches, reflecting the current state of practice in embedded automotive systems.

Within this context, we analyzed the implementation of the Bluetooth stack, a key entry point in modern infotainment systems. We show that this stack remains vulnerable to `CVE-2018-20378`, despite vendor claims that the embedded version falls outside the affected range. Importantly, the original exploitation assumptions of this vulnerability do not hold in our target system, preventing a direct reuse of prior techniques.

Nevertheless, we demonstrate that the vulnerability remains exploitable by developing adapted exploitation primitives, ultimately achieving **remote code execution without user interaction** despite standard mitigations such as ASLR and DEP.

Our results highlight a gap between vulnerability specifications and their practical exploitability in real-world embedded systems. In particular, they show that version-based assumptions and existing mitigations do not necessarily reflect the actual security posture of deployed platforms.

A responsible disclosure process was conducted with the manufacturer, enabling remediation across multiple vehicle models.

References

[1] M. Evdokimov, "0-click RCE on the IVI component: Pwn2Own Automotive edition," in *Hexacon*, Paris, France, october 2024.

[2] CyberThreat Research Lab, "Under Pressure: Exploring a Zero-Click RCE Vulnerability in Tesla's TPMS," december 2024, `https://vicone.com/blog/`.

[3] D. Antonioli and M. Payer, "On the Insecurity of Vehicles Against Protocol-Level Bluetooth Threats," in *43rd IEEE Security and Privacy, SP Workshops*. IEEE, 5 2022, pp. 353–362.

[4] V. Renganathan, E. Yurtsever, Q. Ahmed, and A. Yener, "Valet attack on privacy: a cybersecurity threat in automotive Bluetooth infotainment systems," *Cybersecurity*, vol. 5, no. 1, 2022.

[5] K. Tindell, "CAN Injection: keyless car theft," march 2023, https://kentindell.github.io/2023/04/03/can-injection/.

[6] C. Miller and C. Valasek, "Remote Exploitation of an Unaltered Passenger Vehicle," *DEF CON*, august 2013.

[7] S. Dudek, J.-C. Delaunay, and V. Fargues, "V2G Injector: Whispering to cars and charging units through the Power-Line," in *Actes du Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC)*, Rennes, France, june 2019. [Online]. Available: https://www.sstic.org/2019/presentation/v2g_injector_playing_with_electric_cars_and_charging_stations_via_powerline/

[8] J. Petit, B. Stottelaar, M. Feiri, and F. Kargl, "Remote Attacks on Automated Vehicles Sensors: Experiments on Camera and LiDAR," in *BlackHat Europe*, Amsterdam, Netherland, november 2015.

[9] Globalplatform's Automotive Task Force, "Cybersecurity in Automotive," november 2022, https://globalplatform.org/resource-publication/cybersecurity-in-automotive-v1-0/.

[10] Mikhail Evdokimov, "PerfektBlue - critical vulnerabilities in OpenSynergy Blue SDK," july 2025, https://perfektblue.pcacybersecurity.com/.

[11] B. Caspi, "CVE-2018-20378," march 2019, https://nvd.nist.gov/vuln/detail/CVE-2018-20378.

[12] Bluetooth Core Specification Working Group, *Bluetooth Core Specification*, Std., july 2021.

Appendix

1. Bluetooth Protocol

The Bluetooth protocol stack [12] is a modular software architecture that supports audio, video, file transfer, and Internet tethering. It is organized into distinct layers that separate hardware and software concerns, as illustrated in Figure 3. The lower layers, including the physical layer (RF) and the controller (LMP and LL), jointly referred to as the *Modem* or *Controller*, handle radio-frequency transmissions and low-level link mechanisms. The upper layers, referred to as the *Host*, implement higher-level functions such as connection establishment, authentication, and application profiles. The interface between controller and host is the HCI. This layering ensures interoperability across vendors and has driven widespread adoption.

Within the *Host* stack, L2CAP plays a central role by multiplexing upper-layer protocols such as SDP and RFCOMM and adapting them for efficient transport over HCI. It configures communication channels according to the needs of application protocols, handling fragmentation, segmentation, and multiplexing.

SDP is a client-server protocol used in Bluetooth communications to *discover* services offered by a peer device. Discovery can occur pre-pairing: any nearby device, including those never seen before, may issue SDP

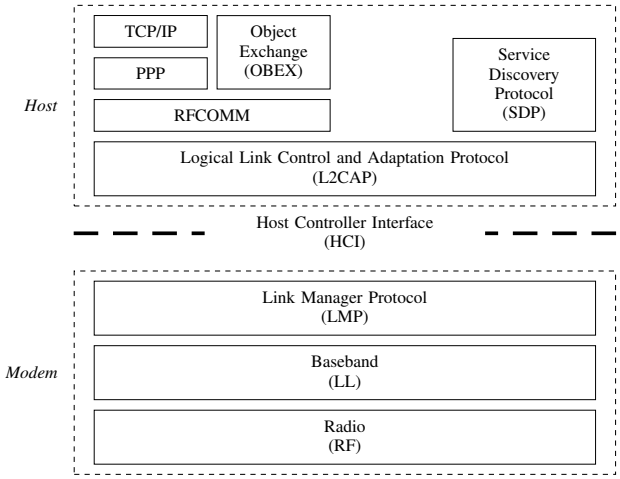

Figure 3. Simplified Bluetooth protocol stack.

queries, and no user approval is required for the querying peer. SDP streamlines connection setup by providing information on available services, including characteristics, attributes, and endpoints. Each service is described by a service record consisting of attributes (key-value pairs) such as the service Universally Unique IDentifier (UUID), communication parameters, and other metadata. Figure 4 shows an example SDP exchange between a phone and a headset.

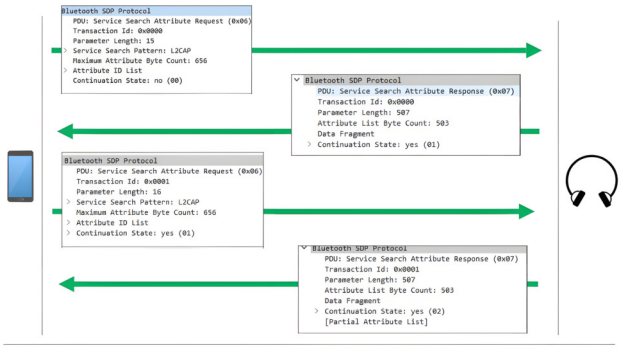

Figure 4. Example SDP exchange between a client (phone) and a server (headset), inspired by [11]. This pre-pairing exchange informs the phone that the peer is an audio device, the presence of buttons and a microphone, and other capabilities.

To enumerate services supported by a Bluetooth device, the Bluetooth standard, in the section SDP, defines two primary request types:

- **Service search**: The client initiates a `ServiceSearchRequest` specifying the UUIDs of the desired service classes [12]. The server responds with a `ServiceSearchResponse`, which contains a list of matching service record handles. These handles can then be used by the client for further inspection or queries.
- **Service attribute retrieval**: To obtain attributes for a specific service, the client issues a `ServiceAttributeRequest` with the previously returned handle and a list of desired attributes. The server replies with

a ServiceAttributeResponse containing key-value pairs for the requested attributes, which clients use to select and connect to suitable services.

To reduce bandwidth consumption, clients often use the combined ServiceSearchAttributeRequest, which merges both previous queries. We focus on this request in the remainder of the paper because it allows us to obtain the largest responses. In Figure 4, the exchange uses ServiceSearchAttributeRequest and ServiceSearchAttributeResponse package types.

The ServiceSearchAttributeRequest contains two fields of particular interest here:

- Maximum Attribute Byte Count: In addition to the receive window configured via the L2CAP channel MTU, this field specifies the maximum number of attribute bytes the client is willing to receive in a single response. It ensures that server responses do not exceed the client's processing or memory capacity.
- Continuation State: Used when response data exceed the Maximum Attribute Byte Count or the L2CAP MTU. It enables partial responses and subsequent retrievals by carrying an identifier that the client must echo to obtain the remaining data, ensuring ordered fragmentation. The Continuation State comprises two fields: count and continuation_state_info. The count field indicates the size (in bytes) of continuation_state_info, which is an opaque byte sequence that the client must resend in the next request to receive the next fragment.

2. Linux ASLR

When a program starts, the execve syscall triggers the kernel routine load_elf_binary, which ultimately maps the executable and performs address randomization for its segments. In our case, libc.so is mapped by the dynamic loader ld_linux into the shared-library region.

The kernel sources[2] show that the TASK_SIZE macro defines the maximum userland address range based on CONFIG_DRAM_SIZE. On our target, TASK_SIZE is 0x80000000 (2 GB). Userland thus spans addresses $[0, \texttt{TASK\_SIZE} - 1]$. Shared objects are loaded starting from a nominal base 0x40000000 plus a random offset, and additional libraries are placed contiguously thereafter.

The libbluego.so library is handled slightly differently. It is loaded at runtime by the Java application bluego via Android's Native Development Kit (NDK), using System.load. Mappings created this way reside in a private heap area distinct from the regular shared-library region, so their base randomization is independent of the standard 0x40000000-based scheme. In our setup, the libbluego.so code segment is mapped around 0x5b000000, and the kernel routine arch_randomize_brk (Listing 5) contributes additional entropy.

2. Available at: https://github.com/boundarydevices/linux.git

```
514 unsigned long arch_randomize_brk(struct mm_struct *mm)
515 {
516   unsigned long range_end = mm->brk + 0x02000000;
517   return randomize_range(mm->brk, range_end, 0) ? :
    ↪   mm->brk;
518 }
```

Listing 5: Function arch_randomize_brk in arch/arm/kernel/process.c, Linux kernel 3.0.35.

To defeat ASLR, we must recover the random offsets for the code segment, the shared-library base, and the heap. From Listing 5, line 516, the maximum span for randomization on this architecture is 0x02000000. However, as shown by randomize_range (Listing 6, line 517), this range is page-aligned, which truncates effective entropy.

```
1342 unsigned long
1343 randomize_range(unsigned long start, unsigned long end,
     ↪   unsigned long len)
1344 {
1345   unsigned long range = end - len - start;
1346
1347   if (end <= start + len)
1348     return 0;
1349   return PAGE_ALIGN(get_random_int() % range + start);
1350 }
```

Listing 6: Function randomize_range in drivers/char/random.c, Linux kernel 3.0.35.

The PAGE_ALIGN macro (Listing 6, line 1349) aligns the randomized address to a 4 kB page boundary, i.e., the lowest 12 bits are cleared. Consequently, the effective entropy affects bits 12-25 only, yielding 0x2000 possible offsets (8192 values). While this would be weak in a fast channel, our Bluetooth path is extremely slow (a few bytes per second). Brute-forcing 8192 trials is impractical in our setting.