

Fighting cavities: Securing Android Bluetooth by Red Teaming

2025-05-17



Speakers

Jeong Wook (Matt) Oh (Offensive Security Engineer)

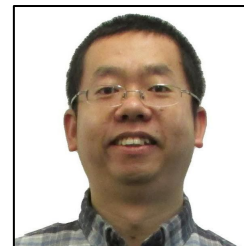
Xuan Xing (Android Red Team Manager)

Rishika Hooda (Offensive Security TPM)

Contributors - Special call out to Bluetooth Feature Team (Brian Delwiche)



Jeong Wook
(Matt) Oh



Xuan Xing



Rishika Hooda

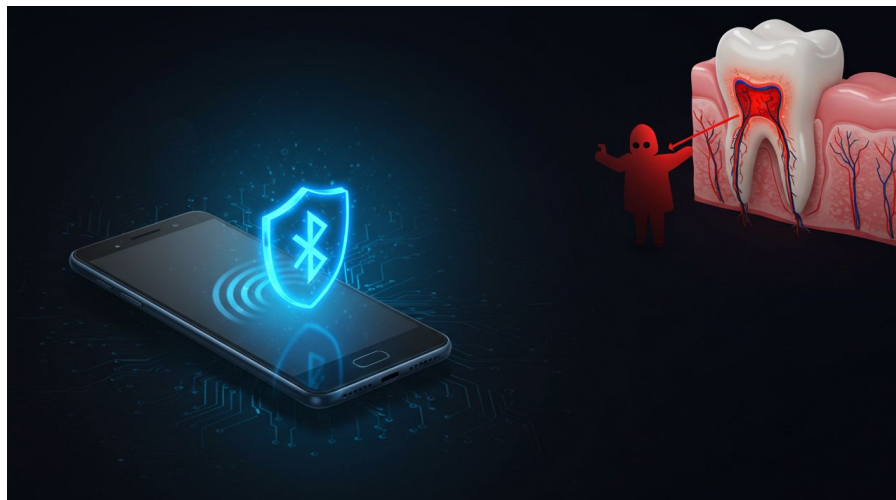
Android Red Team

Mission

Protect the Android ecosystem through offensive security.

Agenda

- Overview
 - Bluetooth Protocol Stack
 - Threat Modelling
- Methodology
 - Code Reviews
 - Fuzzing/CodeQL/KLEE
- Bugs and Exploits
 - GATT Integer Overflow RCE
 - AVDT Type Confusion RCE
- Mitigations & Improvements
- Q&A



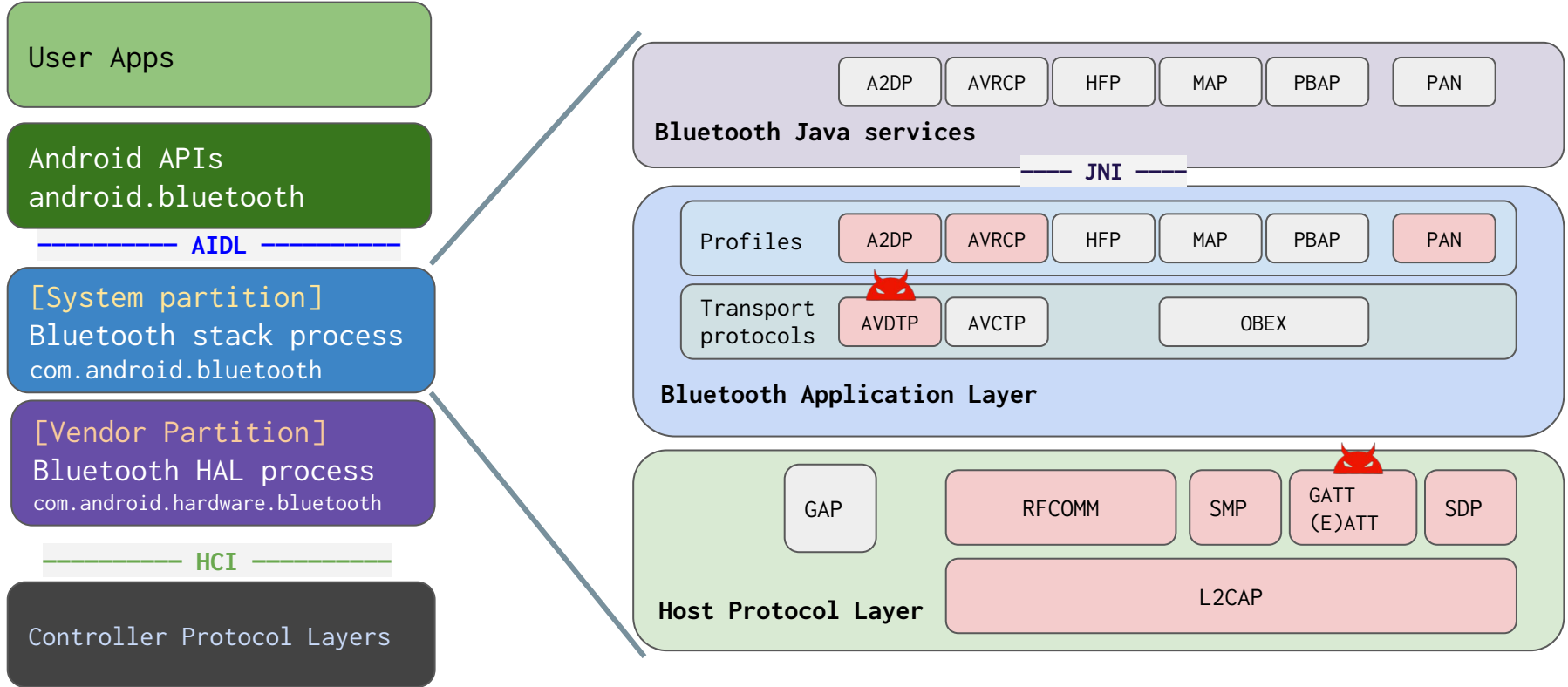


Overview



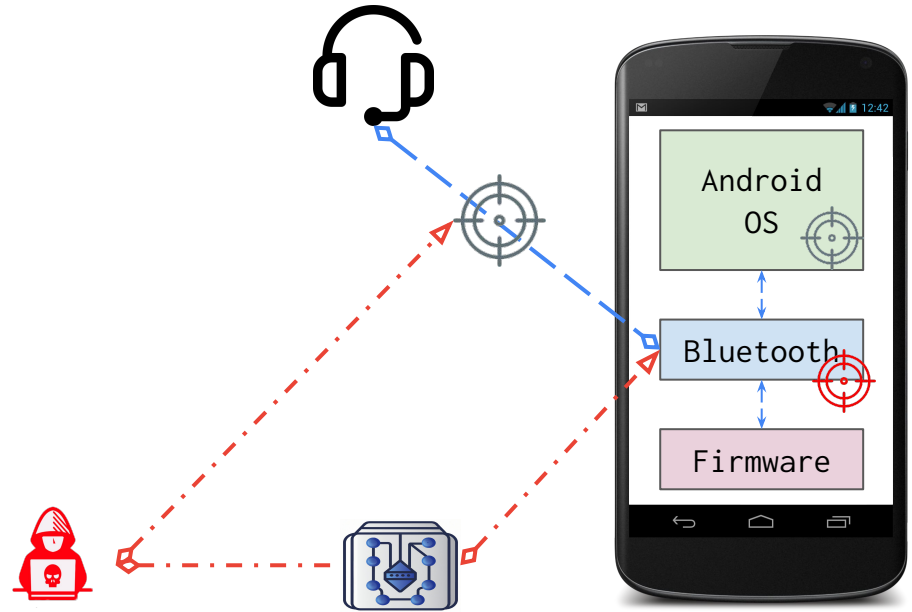
Bluetooth: A Convenient Target in Our Connected World

Bluetooth Protocol Stack (Attacker's view)



Threat Model

- Attacker sending malformed packets, resulting in
 - DoS
 - Information disclosure
 - Code execution
 - Using privileges of the Bluetooth process to affect other parts of Android OS
- Both pre-pairing and post-pairing attacks considered



High Level Impact

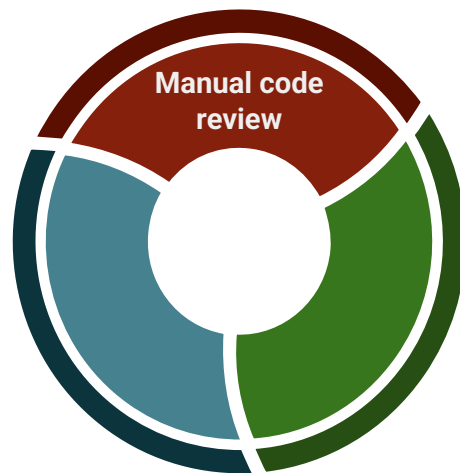
- Discovered multiple Critical/High vulnerabilities
 - All reported Vulns are fixed & patches are delivered via ASB(Android security bulletin)
- Manual code review uncovered bugs that are hard to find with fuzzing
- Experimented KLEE for symbolic execution and found interesting bugs
- Open-source contributions
 - CodeQL, Ubertooth, Wireshark, Scapy and (in-progress) Frida



Methodology

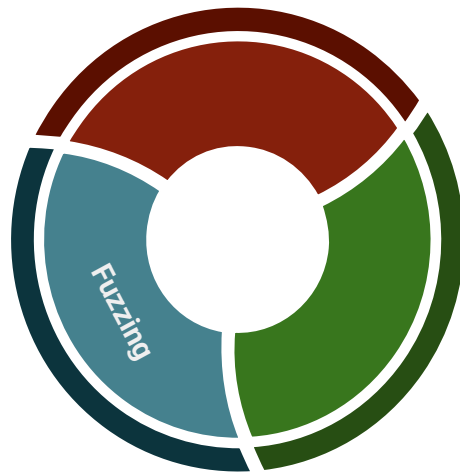
Manual Code Review

- Still the most effective way to discovery high complexity/impactful bugs
 - ~50% of total discoveries
 - Including both bugs we exploited
- Focusing areas hard to automate
 - Logic bugs
 - E2E scenarios involving many components
 - Race conditions
- Limitations
 - Needs domain knowledge
 - One time effort
 - Not scalable



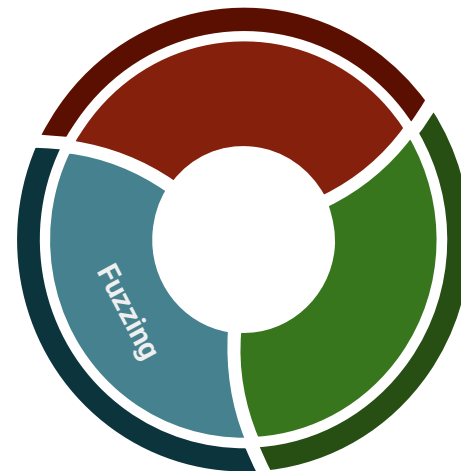
Dynamic Analysis (Fuzzing)

- Scalable, long lasting effect for
 - New issue discovery
 - Regression detection
- New fuzzers created to cover the following areas:
 - SDP, SMP, GATT, BNEP, AVRC, L2CAP
- Existing fuzzers improved to:
 - Support host based fuzzing
- All fuzzers running continuously in Google's internal fuzzing infrastructure



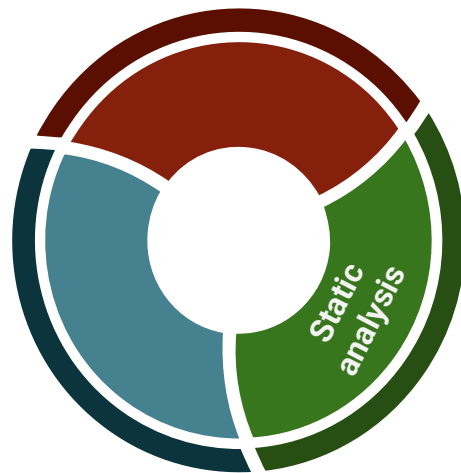
Dynamic Analysis (Fuzzing)

Name	Coverage* %	Discovery
avrc/avct	50%	1
bnep	61%	1
gatt	54%	7
smp	50%	1
sdp	62%	5
l2cap	35%	3



Static Analysis

- **CodeQL**
 - Variant/pattern analysis, for example, OOB array accesses
 - Found new issues, as well as re-found GATT bug exploited in this slides
 - 2 bugs in the CodeQL tool reported and fixed
- **KLEE**
 - Formal verification, an alternative to CBMC
 - Experimented on l2cap, sdp and btm for symbolic execution
 - Found 6 likely OOB read bugs in btm, re-produced another issue
 - Challenges: invasive build changes, path explosion, function pointers





Bugs and Exploits

Bluetooth: Summary of Findings

Proprietary + Need-to-know

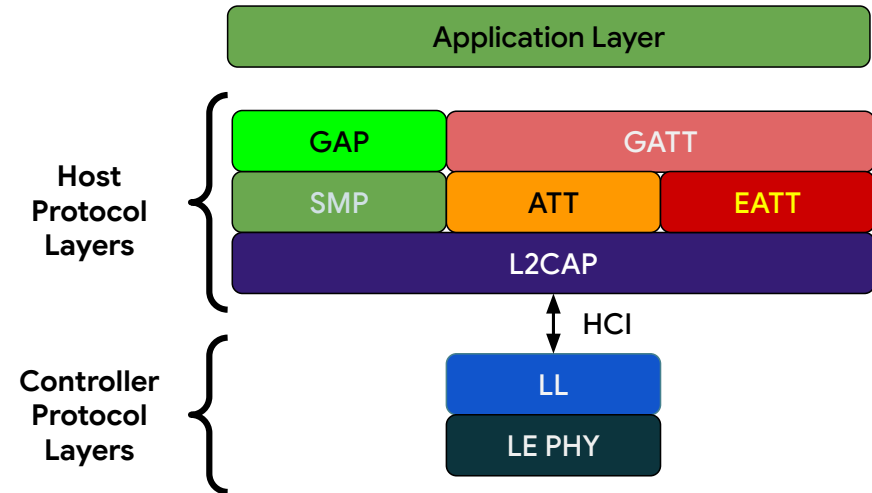
Scope	Component	Critical	High
Application Profiles	A2DP, AVRC, HFP, MAP, PBAP, PAN		CVE-2023-40078
Transport protocols	AVDT, AVCT, OBEX		CVE-2025-22435
Host: other	GAP, ATT/GATT, SMP, RFCOMM, SDP	CVE-2023-35673 CVE-2023-40129 CVE-2023-35681 CVE-2023-35658 CVE-2023-21273	CVE-2024-0045
Core stack implementation	BTA, BTM, BTIF, BTU, BNEP		CVE-2024-0016 CVE-2023-40090 CVE-2023-40087



GATT Integer Overflow RCE

BLE Protocol Stack

- **BLE (Bluetooth Low Energy)**
 - Bluetooth protocol variation to have minimal power consumption to transmit small amounts of data over short distances
- **EATT (Enhanced Attribute Protocol)**
 - Upgraded version of the standard Attribute Protocol (ATT)
 - Concurrent Transactions/Improved Responsiveness & Throughput
 - **MTU Flexibility:** EATT decouples the ATT Maximum Transmission Unit (MTU) from the L2CAP MTU, offering more flexibility
- **GATT (Generic Attribute Profile)**
 - Builds directly on top of ATT or EATT
 - Services and Characteristics



GATT Vulnerability Details

Proprietary + Need-to-know

- **GATT uses EATT**
 - The EATT's TX MTU is set by BLE peer that trigger uint16_t integer overflow/underflow
- **Impact**
 - Proximal Code Execution as Bluetooth Service
- **Bugs & Patches**
 - [CVE-2023-35681](#) - gatts_process_* functions OOB write ([Patch](#))
 - [CVE-2023-35673](#) - build_read_multi_rsp integer overflow ([Patch](#))
 - [CVE-2023-40129](#) - build_read_multi_rsp underflow lead to OOB Write ([Patch](#))
- **Current Status:** All vulnerabilities are remediated & patches have been delivered via ASB

build_read_multi_rsp Overflow (CVE-2023-35673)

Attacker



<-- BLE -->

EATT Connection →
L2CAP_CMD_CREDIT_BASED_CONN_REQ
(mtu: 0xffeb)

Target



```
void eatt_l2cap_reconfig_completed(const RawAddress& bda, uint16_t
    lcid,
                                   bool is_local_cfg,
                                   tL2CAP_LE_CFG_INFO* p_cfg) {
    ...

    EattChannel* channel = find_channel_by_cid(bda, lcid);

    ...

    channel->tx_mtu_ = p_cfg->mtu; <--- channel->tx_mtu_ is set by
    the attacker (0xffeb)
```

build_read_multi_rsp Overflow (CVE-2023-35673)

Attacker



<-- BLE -->

Target



EATT Connection →

L2CAP_CMD_CREDIT_BASED_CONN_REQ
(mtu: 0xffeb)

GATT Client Request →

GATT_REQ_READ_MULTI (0x0E)

```
void gatt_server_handle_client_req(tGATT_TCB& tcb, uint16_t cid,
                                   uint8_t op_code, uint16_t len,
                                   uint8_t* p_data) {
    ...

    case GATT_REQ_READ_MULTI:
    case GATT_REQ_READ_MULTI_VAR:
        gatt_process_read_multi_req(tcb, cid, op_code, len, p_data);
    ...

void gatt_process_read_multi_req(tGATT_TCB& tcb, uint16_t cid, uint8_t op_code,
                                 uint16_t len, uint8_t* p_data) {

    ...

    if (err == GATT_SUCCESS) {
        trans_id = gatt_sr_enqueue_cmd(tcb, cid, op_code, multi_req->handles[0]); ←
        Read Request is queued
    }
}
```

build_read_multi_rsp Overflow (CVE-2023-35673)

Attacker



<-- BLE -->

Target



EATT Connection →

L2CAP_CMD_CREDIT_BASED_CONN_REQ
(mtu: 0xffeb)

GATT Client Request →

GATT_REQ_READ_MULTI (0x0E)

←GATT Response Build

build_read_multi_rsp (Integer
Overflow) called from
gatt_process_read_multi_req

```
static void build_read_multi_rsp(tGATT_SR_CMD* p_cmd, uint16_t mtu) {
    uint16_t ii, total_len, len;
    ...

    len = sizeof(BT_HDR) + L2CAP_MIN_OFFSET + mtu; <--- Integer Overflow
    BT_HDR* p_buf = (BT_HDR*)osi_calloc(len);
    p_buf->offset = L2CAP_MIN_OFFSET;
    p = (uint8_t*)(p_buf + 1) + p_buf->offset;
    ...
    for (ii = 0; ii < p_cmd->multi_req.num_handles; ii++) {
        ...
        total_len = (p_buf->len + p_rsp->attr_value.len);

        if (total_len > mtu) {
            ...
        } else {
            len = p_rsp->attr_value.len;
        }
        ...
        memcpy(p, p_rsp->attr_value.value, len); <--- Heap Overflow
    }
}
```

gatts_process_* Integer Overflow (CVE-2023-35681)

Attacker



<-- BLE -->

Target



EATT Connection →

L2CAP_CMD_CREDIT_BASED_CONN_REQ
(mtu: 0xffeb)

GATT Client Request →

GATT_REQ_READ_BY_GRP_TYPE (0x10)

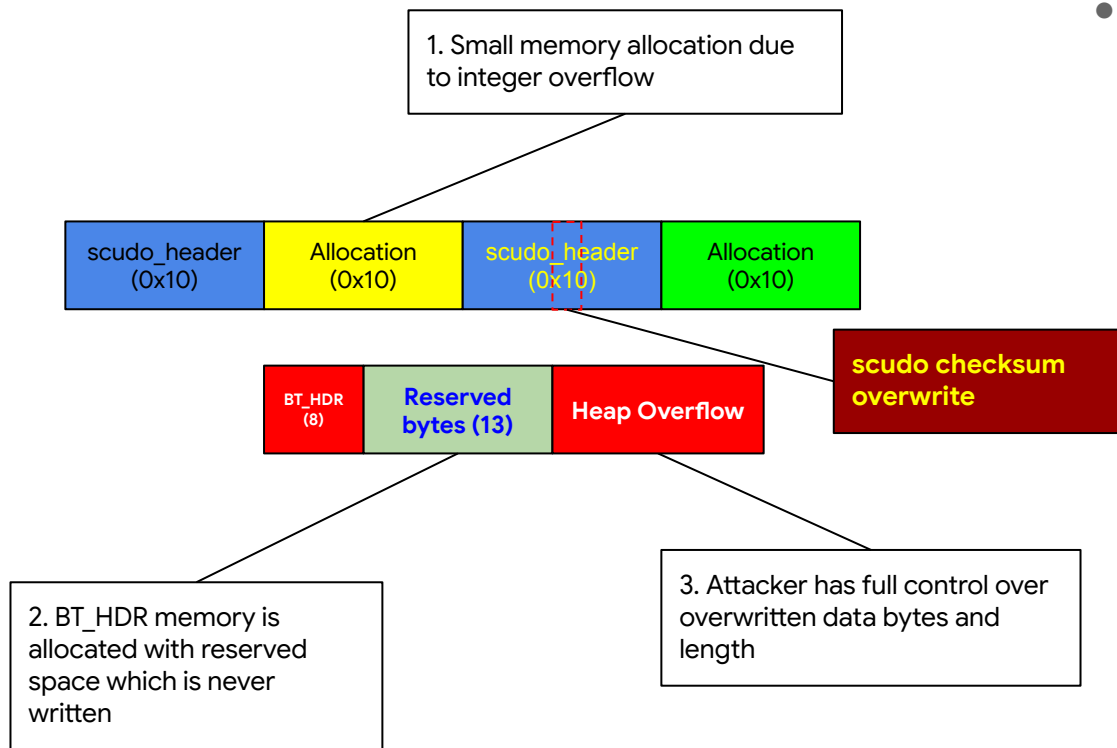
←GATT Handler

gatts_process_primary_service_req
(Integer Overflow)

```
uint16_t gatt_tcb_get_payload_size_tx(tGATT_TCB& tcb, uint16_t cid) {  
    if (!tcb.eatt || (cid == tcb.att_lcid)) return tcb.payload_size;  
  
    EattChannel* channel =  
        EattExtension::GetInstance()->FindEattChannelByCid(tcb.peer_bda, cid);  
  
    return channel->tx_mtu_; <---- Attacker Controllable (64 ~ 0xffff)  
}
```

```
void gatts_process_primary_service_req(tGATT_TCB& tcb, uint16_t cid,  
    uint8_t op_code, uint16_t len,  
    uint8_t* p_data) {  
  
    ...  
  
    uint16_t payload_size = gatt_tcb_get_payload_size_tx(tcb, cid);  
  
    uint16_t msg_len =  
        (uint16_t)(sizeof(BT_HDR) + payload_size + L2CAP_MIN_OFFSET); <----  
    Integer overflow  
    BT_HDR* p_msg = (BT_HDR*)osi_malloc(msg_len); < Heap allocation with  
    small msg_len size
```

Scudo: how it works



- There were some chances where Scudo could have been bypassed through gap in the memory fill, but it is missed by just 2 bytes (which is CRC checksum location)
 - Scudo can be ineffective if non-linear overwrite is possible
 - But, we were unlucky in this case

Scudo is a dynamic user-mode memory allocator, or heap allocator, designed to be resilient against heap-related vulnerabilities (such as heap-based buffer overflow, use after free, and double free) while maintaining performance.

Scudo Bypass: Minimize scudo_free

- Scudo kicks in only when scudo_free is called
 - scudo_free trigger scudo_header checksum error
- Minimize scudo_free call
 - Most of scudo_free during exploitation is triggered by disconnection of the connection
 - Maintain the attack session as long as possible
 - This will removes most of scudo_free calls (90% ~)

Scudo Bypass: retry

- Even though we hit `scudo_free`, the Bluetooth service will restart
 - The restarted process has same memory layout until reboot
 - With restart GATT components run automatically
- For example, [BlueFrag](#) reported 30-50% success rate, but it was enough to achieve RCE because retry attempt is possible with same memory layout

Scudo memory allocation behavior

- The vulnerability allows us to allocate 0x10 ~ 0x20 bytes of memory
 - Same size classes are allocated in proximity of each other
 - The 0x10 allocation (0x20 including scudo_header) class: smallest memory allocation class with following objects allocations
 - **vptr (pointer to vtable)**: our favorite
 - Overwriting this value gives us **full remote code execution**
 - **semaphore_t**: something to avoid
 - Overwriting this value gives no effect, deadlock or **scudo error**
 - **list_node_t (p_cmd->multi_rsp_q)**: useful
 - Overwriting this value gives **scudo error** or **remote OOB Read**

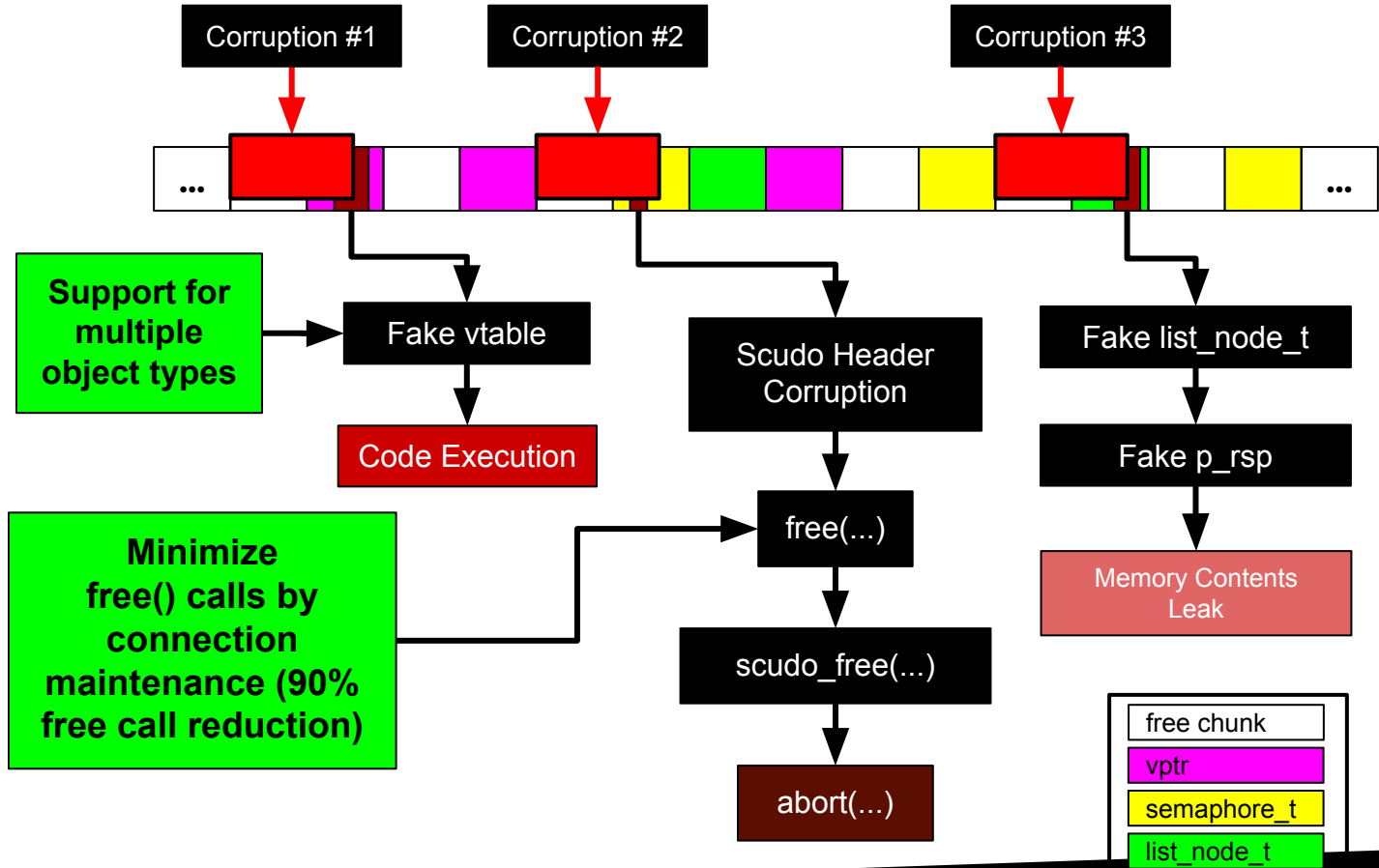
ASLR Bypass: list_node_t (p_cmd->multi_rsp_q)

- OOB Write issue could have used to corrupt `list_node_t` object which has a pointer to `tGATTS_RSP` that is sent to the attacker
- It can read arbitrary memory address with specific patterns
 - This can be used for ASLR bypass
 - It needs additional research to use it for real world ASLR bypass

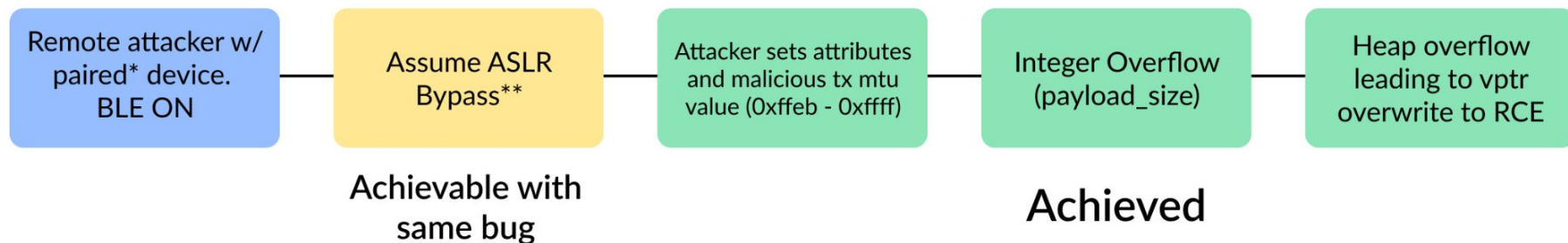
```
static void build_read_multi_rsp(tGATT_SR_CMD* p_cmd,
uint16_t mtu) {
...
    tGATTS_RSP* p_rsp = NULL;
...
    if (node != list_end(list)) p_rsp =
(tGATTS_RSP*)list_node(node); ← list_node_t can be
corrupt and attacker can control p_rsp address
...
    memcpy(p, p_rsp->attr_value.value, len); ←
Attacker can read arbitrary memory address
```

Scudo Bypass: scudo_free vs RCE race

- Increase the success rate
 - Corrupting multiple objects in same session
 - Multiple exploit attempts in same session
 - Minimize corruption footprint: $2 (\text{sizeof}(\text{BT_HDR})) + 8 (\text{scudo_header padding}) + 8 (\text{vptr size})$
- Race
 - By sending multiple malicious requests, the success rate for vptr corruption goes up to **60-70%**



Attack Chain: GATT Integer Overflow



* Device pairing is used for the demo, but MITM with no pairing scenario is possible as the bug happens pre-auth phase

Demo: GATT Integer Overflow

https://youtu.be/E1QFI42_HhU



AVDT Type Confusion RCE

- **AVDTP (Audio/Video Distribution Transport Protocol)**
 - Underlying transport mechanism for high-quality audio/video.
 - Operates above the L2CAP (Logical Link Control and Adaptation Protocol) layer
 - Key Responsibilities:
 - Stream Negotiation: For the audio/video stream - the codec to be used (like SBC, AAC, aptX) and its configuration
 - Connection Management: Setup and teardown of the dedicated channels
 - Multiplexing: Multiple audio/video streams to be managed over a single L2CAP link
 - Data Transport: Actual transfer of the encoded audio or video packets

CVE-2025-22435 - avdt_msg_ind type confusion

Proprietary + Need-to-know

```
void avdt_msg_ind(AvdtPccb* p_ccb, BT_HDR* p_buf) {
    ...
    p_buf = avdt_msg_asmb1(p_ccb, p_buf);
    ...
    p = (uint8_t*)(p_buf + 1) + p_buf->offset;
    /* parse the message header */
    AVDT_MSG_PRS_HDR(p, label, pkt_type, msg_type);
    ...
    /* get and verify signal */
    AVDT_MSG_PRS_SIG(p, sig);
    ...
    /* set up to parse message */
    if ((msg_type == AVDT_MSG_TYPE_RSP) && (sig == AVDT_SIG_DISCOVER)) {
        /* parse discover rsp message to struct supplied by app */
        msg.discover_rsp.p_sep_info = (tAVDT_SEP_INFO*)p_ccb->p_proc_data;
        msg.discover_rsp.num_seps = p_ccb->proc_param;
    } else if ((msg_type == AVDT_MSG_TYPE_RSP) &&
               ((sig == AVDT_SIG_GETCAP) || (sig == AVDT_SIG_GET_ALLCAP))) {
        /* parse discover rsp message to struct supplied by app */
        msg.svccap.p_cfg = (AvdtPsepConfig*)p_ccb->p_proc_data;
    }
}
```

Attacker
Packet

Type Confusion

`p_ccb->p_proc_data` can point to AVDT_SIG_DISCOVER/AVDT_SIG_GET_ALLCAP/AVDT_SIG_GETCAP custom data. If sig is fabricated, it can typecast `tAVDT_SEP_INFO*` to wrong type (`AvdtPsepConfig*`).

Sig is attacker
controllable

- [CVE-2025-22435](#) - avdt_msg_ind type confusion ([Patch](#))

- Type confusion that can lead to heap overflow that leads to remote code execution (proximal)

Fix & Mitigation

- Generic mitigations doesn't work well over type confusion case
- [MTE](#) can mitigate exploitation (Pixel 8+)
 - Armv9 introduced the Arm Memory Tagging Extension (MTE), a hardware extension that allows you to catch use-after-free and buffer-overflow bugs in your native

```
void avdt_msg_ind(AvdtPcb* p_ccb, BT_HDR* p_buf) {
    ...
    avdt_msg_send_grej(p_ccb, sig, &msg);
}
}
+
+ /* validate reject/response against cached sig */
+ if (((msg_type == AVDT_MSG_TYPE_RSP) || (msg_type == AVDT_MSG_TYPE_REJ)) &&
+     (p_ccb->p_curr_cmd == nullptr || p_ccb->p_curr_cmd->event != sig)) {
+     AVDT_TRACE_WARNING(
+         "Dropping msg with mismatched sig; sig=%d", sig);
+     ok = false;
+ }
+ }
}

if (ok && !gen_rej) {
```

Discovery Request

Attacker



← Stream Setup →

Target

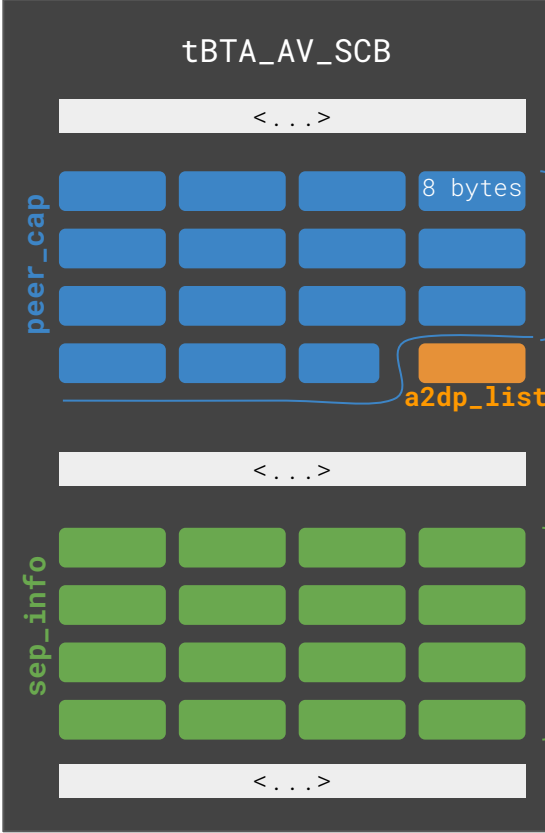


← AVDT_SIG_DISCOVER REQ

AvdtpCcb* p_ccb



```
struct tBTA_AV_SCB {
    ...
    AvdtpSepConfig peer_cap;
    list_t* a2dp_list;
    ...
    tAVDT_SEP_INFO sep_info[BTA_AV_NUM_SEPS];
    ...
}
```



Discovery Response

Attacker



← Stream Setup →

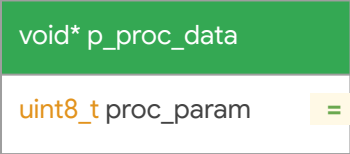
Target



← AVDT_SIG_DISCOVER REQ

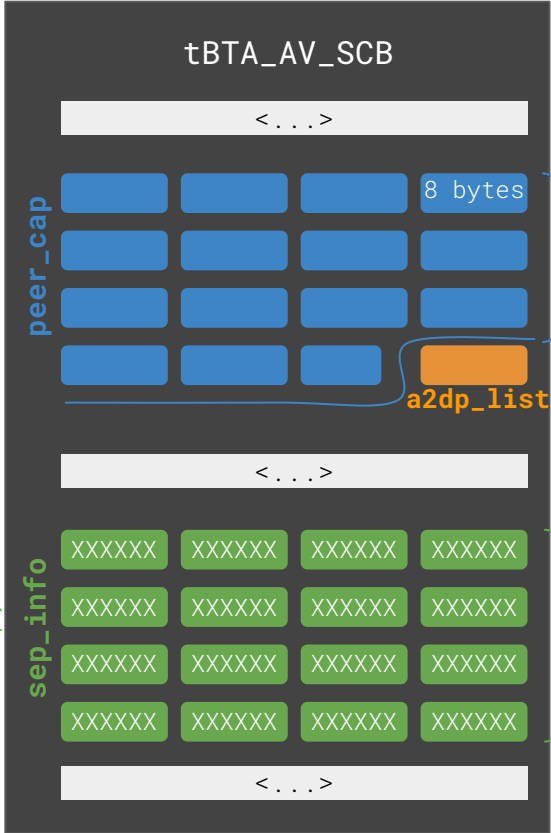
AVDT_SIG_DISCOVER RSP →

AvdtpCcb* p_ccb



```
if ((msg_type == AVDT_MSG_TYPE_RSP) && (sig == AVDT_SIG_DISCOVER)) {
    msg.discover_rsp.p_sep_info = (tAVDT_SEP_INFO*)p_ccb->p_proc_data;
    msg.discover_rsp.num_seps = p_ccb->proc_param;= 32
}

// Call avdt_msg_prs_discover_rsp(&msg, ...)
(*avdt_msg_prs_rsp[AVDT_SIG_DISCOVER])(&msg, p, p_buf->len);
```



Get Capability Request

Attacker



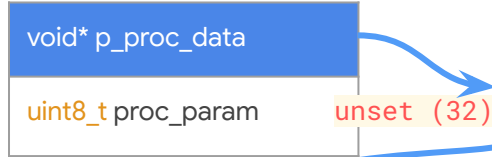
← Stream Setup →

Target

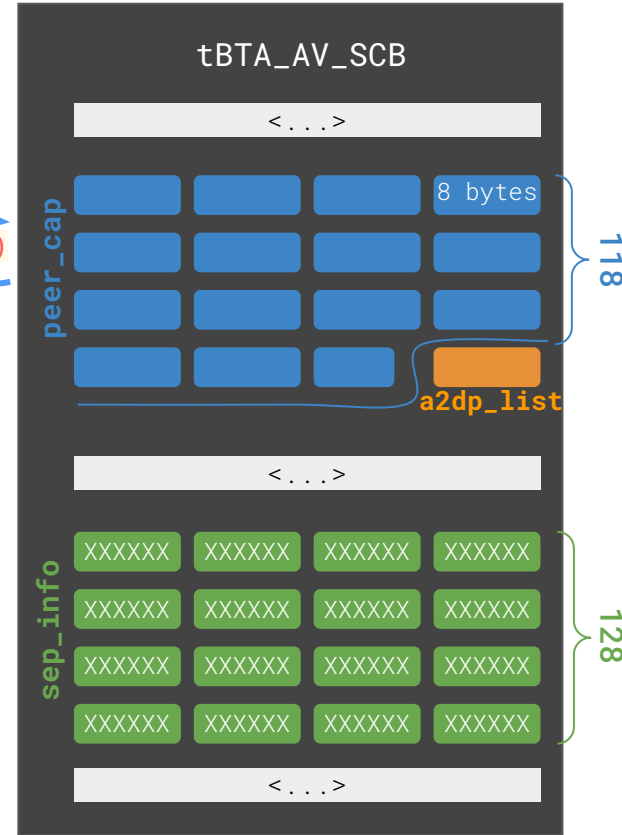


← AVDT_SIG_DISCOVER REQ
AVDT_SIG_DISCOVER RSP →
← **AVDT_SIG_GET_CAP REQ**

AvdtpCcb* p_ccb



```
struct tBTA_AV_SCB {
    ...
    AvdtpSepConfig peer_cap;
    list_t* a2dp_list;
    ...
    tAVDT_SEP_INFO sep_info[BTA_AV_NUM_SEPS];
    ...
}
```



Discovery Response

Attacker



← Stream Setup →

Target



← AVDT_SIG_DISCOVER REQ

AVDT_SIG_DISCOVER RSP →

← AVDT_SIG_GET_CAP REQ

AVDT_SIG_DISCOVER RSP →

AvdtpCcb* p_ccb

void* p_proc_data

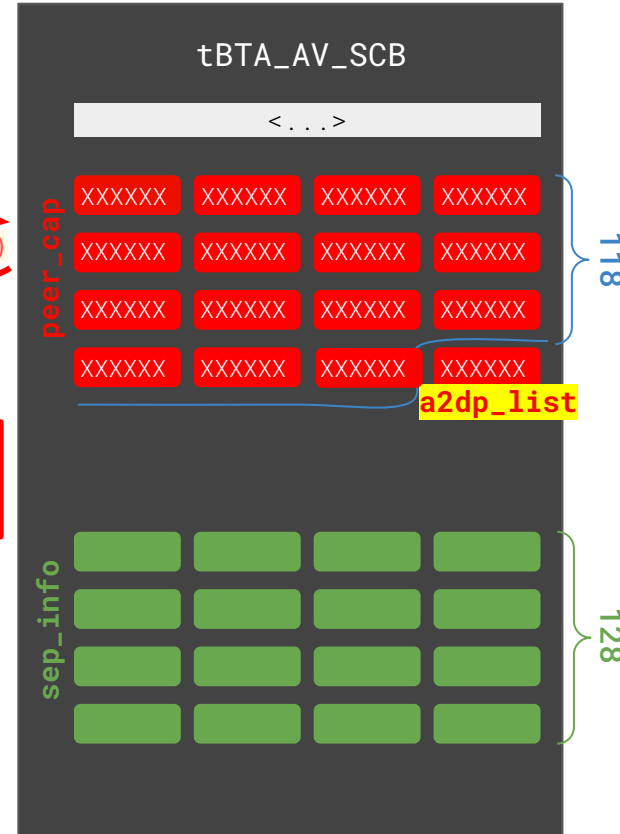
uint8_t proc_param

unset (32)

Type Confusion

The p_ccb->p_proc_data
(peer_cap) is casted into
p_sep_info

```
if ((msg_type == AVDT_MSG_TYPE_RSP) && (sig == AVDT_SIG_DISCOVER)) {  
    msg.discover_rsp.p_sep_info = (tAVDT_SEP_INFO*)p_ccb->p_proc_data;  
    msg.discover_rsp.num_seps = p_ccb->proc_param; unset (32)  
}  
  
// Call avdt_msg_prs_discover_rsp(&msg, ...)  
(*avdt_msg_prs_rsp[AVDT_SIG_DISCOVER])(&msg, p, p_buf->len);
```



Overwrite a2dp_list

```
list_t* a2dp_list
```

```
struct list_t {  
    list_node_t* head;  
    list_node_t* tail;  
    size_t length;  
    list_free_cb free_cb;  
    const allocator_t* allocator;  
};
```

```
struct list_node_t {  
    struct list_node_t* next;  
    void* data;  
};
```

1. Heap spray for predictable addresses to bypass ASLR
2. system(...) gadget
 - a. Fake list structure with free_cb pointing to system
 - b. list->free_cb(head->node->data)
3. Set a2dp_list = 0x4000020000
4. Send packets with the fake structure so we get lucky and it puts our payload at this address
5. When freeing a node, our list at 0x4000020000 will be used, and our free_cb (system) will be called

Heap Spray Data

`list_t* a2dp_list`

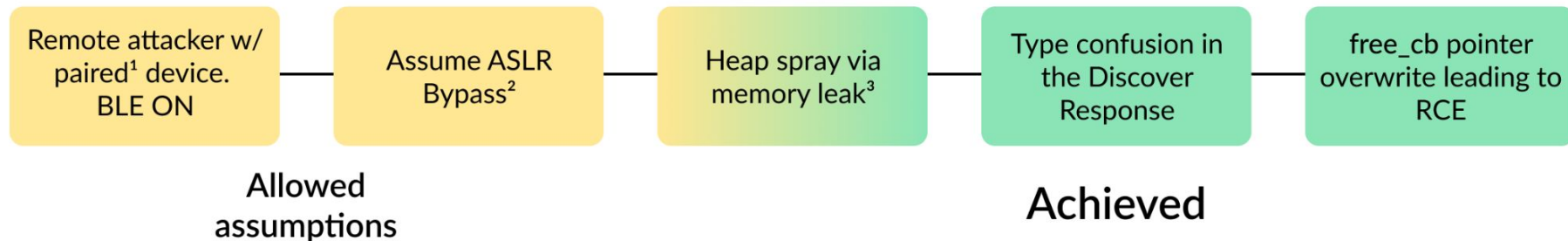
```
struct list_t {  
    list_node_t* head;  
    list_node_t* tail;  
    size_t length;  
    list_free_cb free_cb;  
    const allocator_t* allocator;  
};
```

```
struct list_node_t {  
    struct list_node_t* next;  
    void* data;  
};
```

Heap spray object

<code>0x<...>00</code>	head	= <code>0x<...>28</code>
<code>0x<...>08</code>	tail	= 'AAAAAAAA'
<code>0x<...>10</code>	length	= 1
<code>0x<...>18</code>	free_cb	= system
<code>0x<...>20</code>	allocator	= 'BBBBBBBB'
<code>0x<...>28</code>	next	= 'CCCCCCCC'
<code>0x<...>30</code>	data	= <code>0x<...>38</code>
<code>0x<...>38</code>	"nc -l -p 1337 sh"	

Attack Chain



¹ Device is either already paired or expected to be paired with

² Fair assumption given many "OOB Read" bugs discovered

³ Removed one `free` call to achieve

Demo: AVDT Type Confusion Exploit

<https://youtu.be/E8ZkWkntclg>



Mitigations & Future Improvements

Remediation Challenges

```
179 180
180 181     if (p_rsp != NULL) {
181 182         total_len = (p_buf->len + p_rsp->attr_value.len);
182 183         total_len = p_buf->len;
183 184         if (p_cmd->multi_req.variable_len) {
184 185             total_len += 2;
185 186         }
186 187     }
187 188     if (total_len > mtu) {
188 189         /* just send the partial response for the overflow case */
189 190         len = p_rsp->attr_value.len - (total_len - mtu);
190 191         VLOG(1) << "Buffer space not enough for this data item, skipping";
191 192         break;
192 193     }
193 194     len = std::min((size_t) p_rsp->attr_value.len, mtu - total_len);
194 195     if (len == 0) {
195 196         VLOG(1) << "Buffer space not enough for this data item, skipping";
196 197         break;
197 198     }
198 199     if (len < p_rsp->attr_value.len) {
199 200         is_overflow = true;
200 201         VLOG(1) << StringPrintf(
201 202             "multi read overflow available len=%zu val_len=%d", len,
202 203             p_rsp->attr_value.len);
203 204     } else {
204 205         len = p_rsp->attr_value.len;
205 206     }
206 207     if (p_cmd->multi_req.variable_len) {
207 208         UINT16_TO_STREAM(p, len);
208 209         UINT16_TO_STREAM(p, (uint16_t) len);
209 210     }
210 211     if (p_rsp->attr_value.handle == p_cmd->multi_req.handles[i1]) {
211 212         // check for possible integer overflow
212 213         if (p_buf->len + len <= UINT16_MAX) {
213 214             memcpy(p, p_rsp->attr_value.value, len);
214 215             if (!is_overflow) p += len;
215 216             p_buf->len += len;
216 217         } else {
217 218             p_cmd->status = GATT_NOT_FOUND;
218 219             break;
219 220         }
220 221     }
221 222     ARRAY_TO_STREAM(p, p_rsp->attr_value.value, (uint16_t) len);
222 223     p_buf->len += (uint16_t) len;
```

- Most BT security issues are straightforward, but...
 - Memory safety or spec compliance?
- Biggest challenge is often **reproducing** the issue
- Use a proof of concept where possible, otherwise time-consuming interop tests may be required
- Straightforward fixes can have less straightforward consequences – testing is essential

Mitigations

```
179 180
180 181     if (p_rsp != NULL) {
181     182         total_len = (p_buf->len + p_rsp->attr_value.len);
182     183         total_len = p_buf->len;
183     184         if (p_cmd->multi_req.variable_len) {
184     185             total_len += 2;
185     186         }
186     187         if (total_len > mtu) {
187     188             /* just send the partial response for the overflow case */
188     189             len = p_rsp->attr_value.len - (total_len - mtu);
189     190             VLOG(1) << "Buffer space not enough for this data item, skipping";
190     191             break;
191     192         }
192     193         len = std::min((size_t) p_rsp->attr_value.len, mtu - total_len);
193     194         if (len == 0) {
194     195             VLOG(1) << "Buffer space not enough for this data item, skipping";
195     196             break;
196     197         }
197     198         if (len < p_rsp->attr_value.len) {
198     199             is_overflow = true;
199     200             VLOG(1) << StringPrintf(
199     201                 "multi read overflow available len=%zu val_len=%d", len,
200     202                 p_rsp->attr_value.len);
200     203         } else {
201     204             len = p_rsp->attr_value.len;
201     205         }
202     206         if (p_cmd->multi_req.variable_len) {
202     207             UINT16_TO_STREAM(p, len);
203     208             p_buf->len += 2;
203     209         }
204     210         if (p_rsp->attr_value.handle == p_cmd->multi_req.handles[ii]) {
204     211             // check for possible integer overflow
205     212             if (p_buf->len + len <= UINT16_MAX) {
205     213                 memcpy(p, p_rsp->attr_value.value, len);
206     214                 if (!is_overflow) p += len;
206     215                 p_buf->len += len;
207     216             } else {
207     217                 p_cmd->status = GATT_NOT_FOUND;
208     218                 break;
208     219             }
209     220         }
209     221         ARRAY_TO_STREAM(p, p_rsp->attr_value.value, (uint16_t) len);
210     222         p_buf->len += (uint16_t) len;
211     223     }
```

- Basic memory safety: check bounds, etc.
- For the AVDT type confusion issue, cache the type and compare
- General mitigations
 - Where possible, bake checks into higher-level constructs
 - Use language idioms or well-tested libraries to move data rather than rolling our own

Hardening Efforts & Path Forward

Proprietary + Need-to-know

- **Proactively finding vulnerabilities via**
 - Static analysis
 - Red Teaming Exercises
 - Better interop tests, run regularly
 - Fuzzing
 - Many of these issues are effectively regressions
- **Refactors for better memory safety, where possible**
 - More compiler mitigations
 - Safer languages, or safer C++ types





Questions? Read our blogs!

