riscure

driving your security forward

black hat EUROPE 2021

november 10-11, 2021

BRIEFINGS

Achieving Linux Kernel Code Execution Through A Malicious USB Device

Martijn Bogaard
Principal Security Analyst
martijn@riscure.com
@jmartijnb

Dana Geist
Senior Security Analyst
@geistdana

Our agenda

- Why USB based attacks?
- The forgotten vulnerability (a.k.a CVE-2016-2384)
- Exploitation approach
- Demo
- Discussion / Final thoughts

Why USB based attacks?

Why USB based attacks?

- Can be found in almost every device
- Accessible from early boot (ROM) and during runtime
- Complex protocol == potential for vulnerabilities
- Recent attacks in the wild demonstrated its potential

The forgotten vulnerability

CVE-2016-2384

CVE-2016-2384: what is it about?

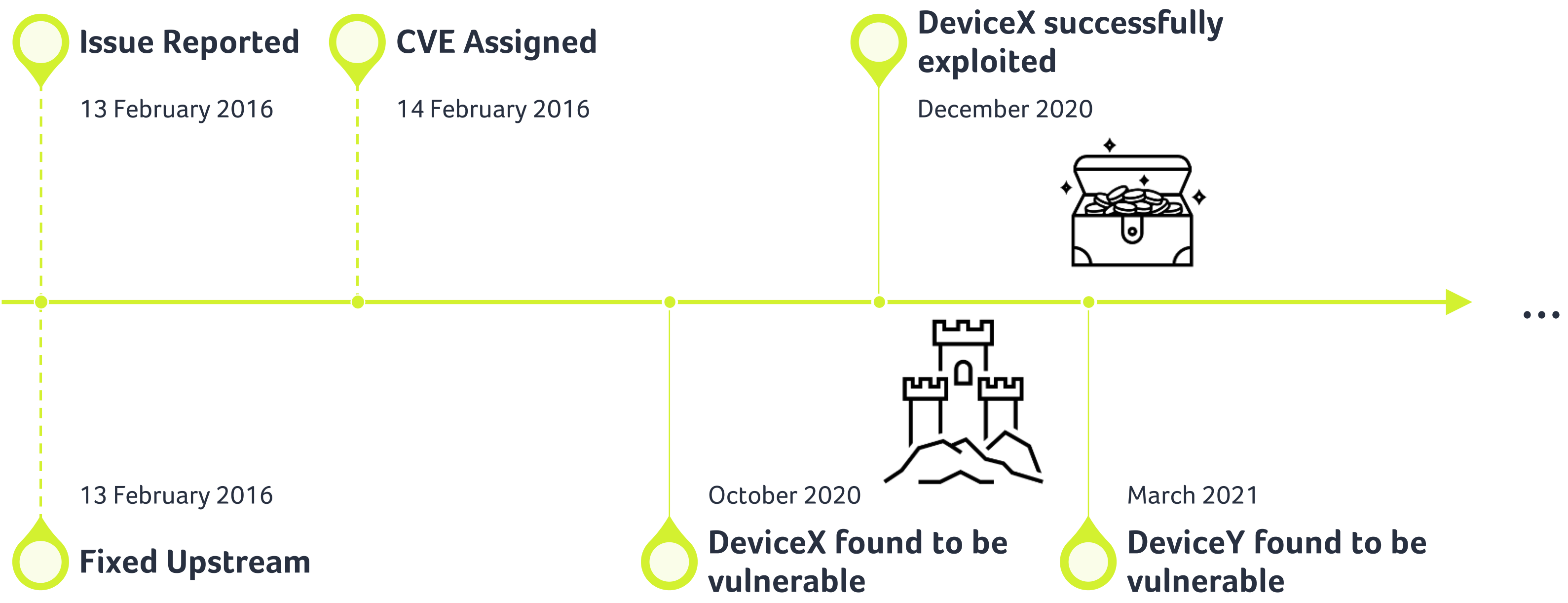
The forgotten vulnerability

- **Double free** in the Linux kernel found[1] by **Andrey Konovalov**
- Targets the USB MIDI subsystem
- PoCs demonstrating:
  - Denial of Service
  - Code execution in the kernel turning the double-free into a use-after-free by:
    - Unprivileged code execution (i.e.: syscall interface)
    - Sockets (i.e.: allocation of SKBs)
- Affected major Linux based systems and distributions such as Ubuntu Fedora and CentOS assuming physical access

[1] https://xairy.io/articles/2016/cve-2016-2384

CVE-2016-2384: why do we still care?

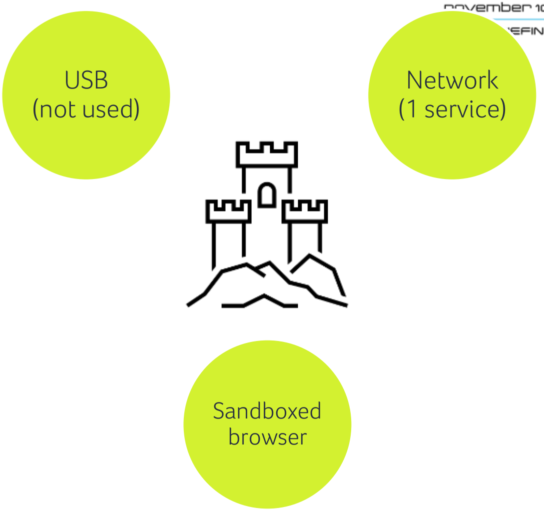
The forgotten vulnerability



DeviceX characteristics

The forgotten vulnerability

- Widely used device
- Very minimal Linux-based system
  - Kernel early 4.x
- Actively backporting security fixes
- Locked down (attack surface reduction)
  - No crash logs, no serial output

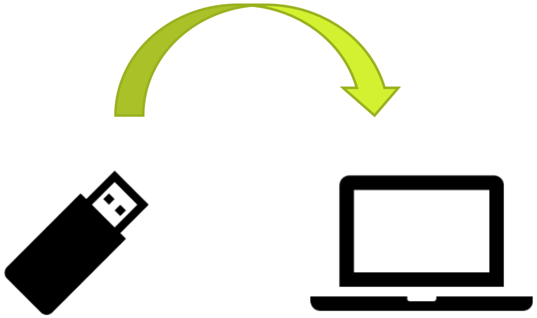


The actual device is not relevant for this talk ;-)

CVE-2016-2384: how is the double-free triggered?

The forgotten vulnerability

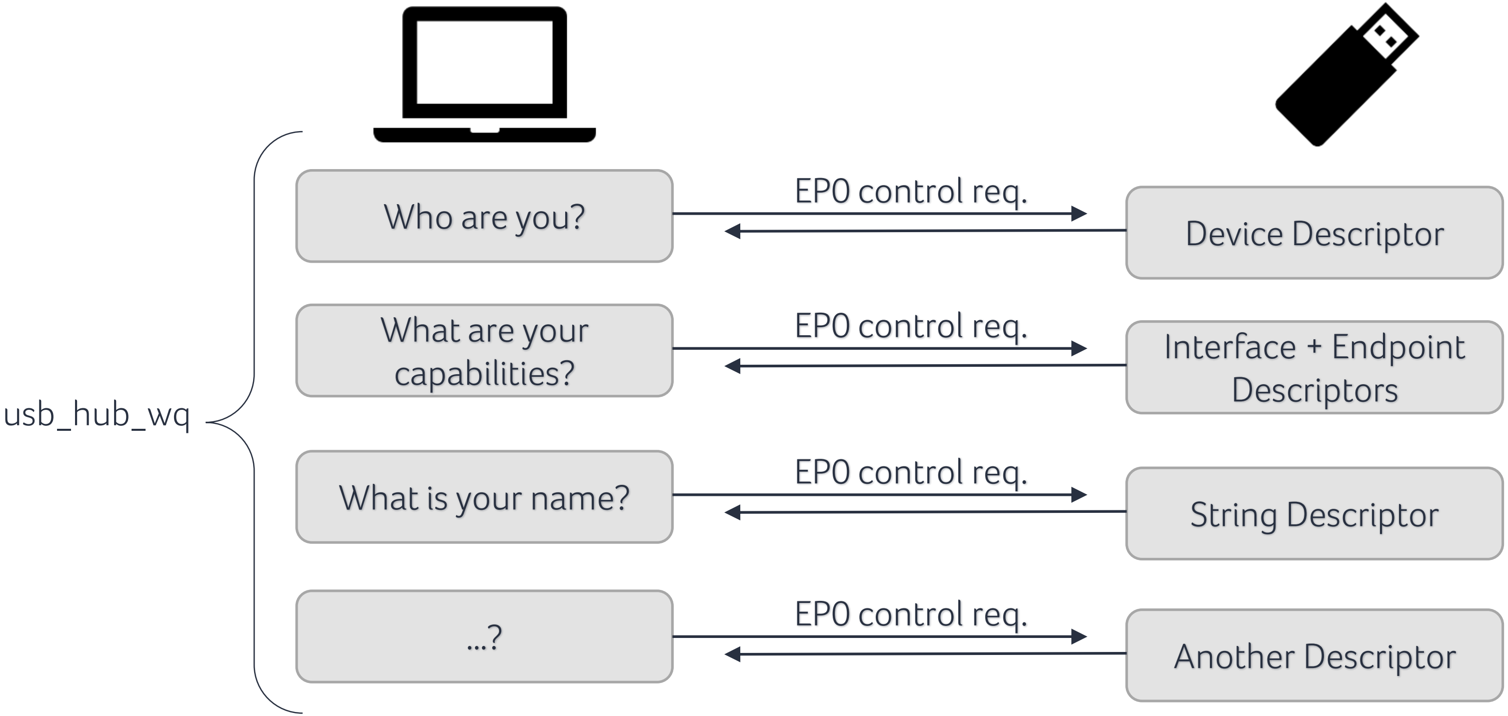
- A USB MIDI device is connected to the target device
  - The device's configuration is not standard



USB probing process

USB crash course

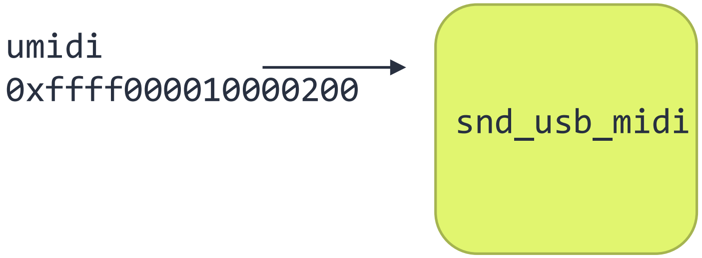
During probing all communication is initiated by the host:



CVE-2016-2384: how is the double-free triggered?

The forgotten vulnerability

- Since the device is of type MIDI, a specific function is called: `__snd_usbmidi_create()`
- The function allocates a structure on the heap



CVE-2016-2384: how is the double-free triggered?

The forgotten vulnerability

- A (deliberate) error in the MIDI configuration is encountered:
  - Causing the 1st free: `snd_usbmidi_free(umidi)`
  - But also returns an error which makes the entire probing process fail
- As part of the cleanup process a free is executed on the same memory location
  - Causing the 2nd free: `snd_usbmidi_free(umidi)`

To SLAB or to SLUB

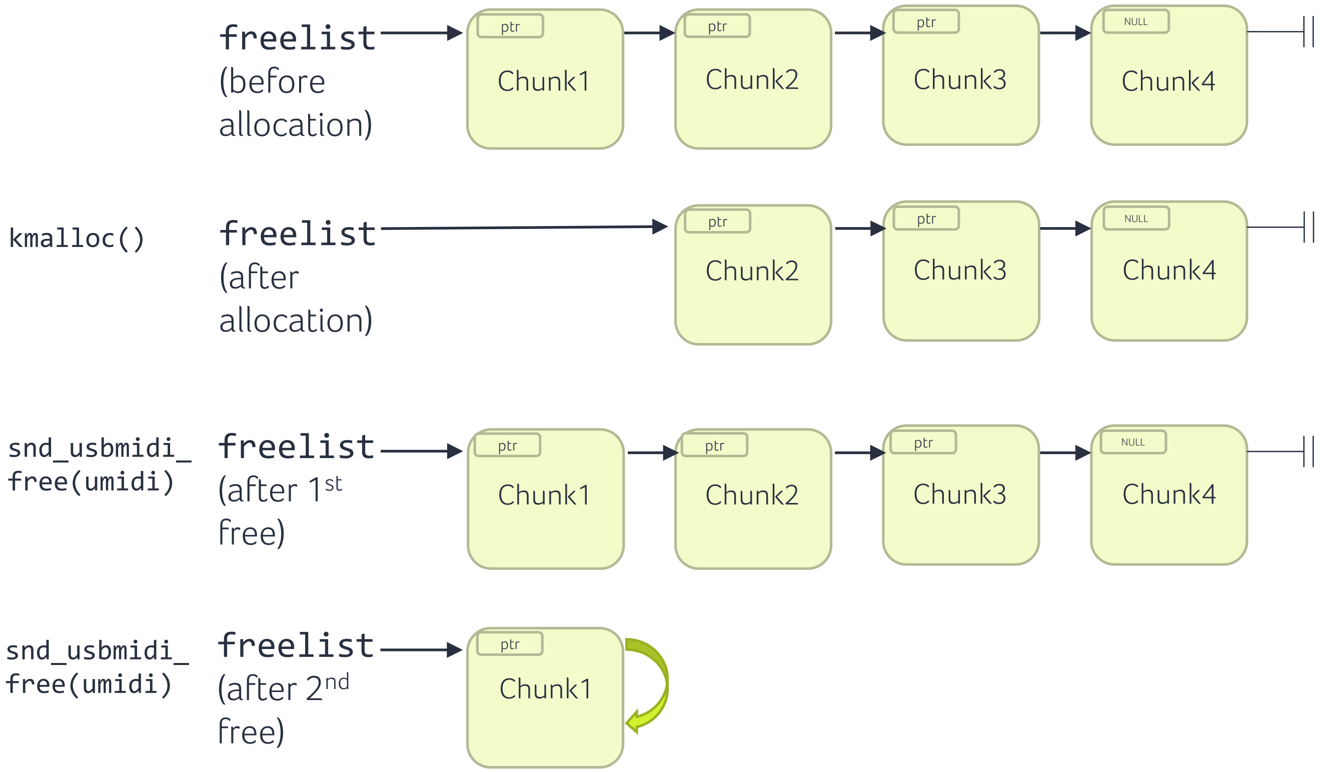
SLUB allocator

- SLUB is the default allocator in Linux (since 2.6.23)
  - Implements the nitty gritty details of the kernel allocations and deallocations
- SLUB groups allocated chunks into *slabs*
  - A slab is a collection of objects of the same (rounded) size
  - The freelist is arranged as a simple linked list (next ptr = only metadata)
  - When allocating a new chunk, the first object in the list will be removed from the list and its pointer returned

This specific behavior shapes the freelist in a very specific way after a double-free occurs!

CVE-2016-2384: what happens with the heap?

SLUB allocator



CVE-2016-2384: the fix

The forgotten vulnerability

```
diff --git a/sound/usb/midi.c b/sound/usb/midi.c
index cc39f63299ef0..007cf58311215 100644
--- a/sound/usb/midi.c
+++ b/sound/usb/midi.c
@@ -2455,7 +2455,6 @@ int snd_usbmidi_create(struct snd_card *card,
       else
               err = snd_usbmidi_create_endpoints(umidi, endpoints);
       if (err < 0) {
-              snd_usbmidi_free(umidi);
               return err;
       }
```

Our exploitation approach

USB probing process

USB crash course

- Devices are constrained to reply to the host requests and cannot:
  - Initiate arbitrary communication
  - Send arbitrary data in USB packets

→ Limited exploitation primitives!

The midi object

Exploitation approach

- Let’s have a look at our MIDI object
- Struct size is between 256 and 512
- This means we need to focus on the slab-512
  - Big enough to hold a reasonable payload
  - In general this is a low activity slab

```
umidi → struct snd_usb_midi
        size = 272
```

This increases our chances of winning the race!

Exploitation primitive

Exploitation approach

- So how can turn a double free into something useful?
  - Remember, we have a loop in our freelist
- All allocations on the same CPU + Slab get the same chunk

  → Often results in a kernel panic within seconds

But every allocation updates the freelist ptr with the content of the first 8 bytes of the chunk!

Exploitation primitive

Exploitation approach

- All allocations on the same CPU + Slab get the same chunk

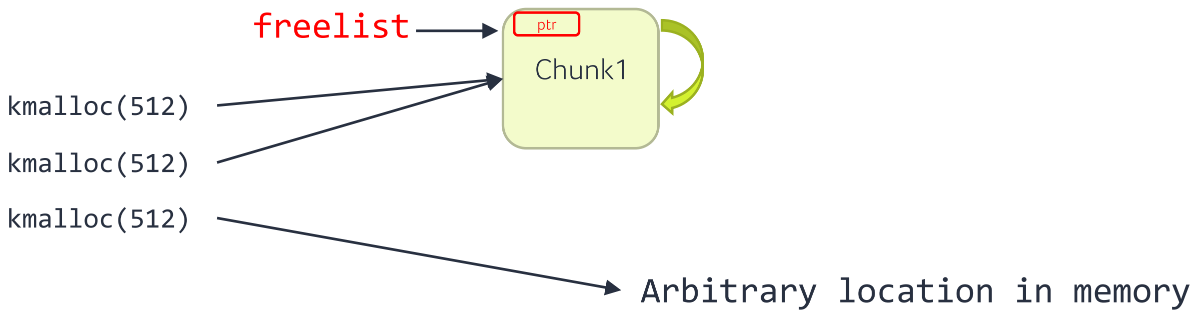
But every allocation updates the freelist ptr with the content of the first 8 bytes of the chunk!

→ 2 consecutive allocations with control over the content will result in freelist ptr control

Exploitation primitive

Exploitation approach

- All allocations on the same CPU + Slab get the same chunk



But every allocation updates the freelist ptr with the content of the first 8 bytes of the chunk!

→ 2 consecutive allocations with control over the content will result in freelist ptr control
→ The 3rd allocation gives then an arbitrary write primitive!

Exploitation primitive

Exploitation approach

- 512-slab chunk
- We need 3 kmallocs in a row
- Controlled data in the first and last allocation
    - 1st chunk: contains pointer to **`arbitrary_mem_location`**
    - 3rd chunk: payload written to **`arbitrary_mem_location`**

So where to find such primitive…?

- Let's dive in another USB driver

USB HID

Exploitation approach

- Human Interface Device protocol
  - A generic protocol for keyboards, mouse's, game controllers, etc.
  - Describes a device as a series of inputs and outputs
- Uses HID and HID_REPORT descriptors to describe the device functions
  - Report descriptor is a variable length blob of data (up to 4 KiB)

```
0x05, 0x01,          // USAGE_PAGE (Generic Desktop)
0x09, 0x02,          // USAGE (Mouse)
0xa1, 0x01,          // COLLECTION (Application)
0x09, 0x01,          //   USAGE (Pointer)
0xa1, 0x00,          //   COLLECTION (Physical)
0x05, 0x09,          //     USAGE_PAGE (Button)
0x19, 0x01,          //     USAGE_MINIMUM (Button 1)
0x29, 0x03,          //     USAGE_MAXIMUM (Button 3)
0x15, 0x00,          //     LOGICAL_MINIMUM (0)
0x25, 0x01,          //     LOGICAL_MAXIMUM (1)
0x95, 0x03,          //     REPORT_COUNT (3)
0x75, 0x01,          //     REPORT_SIZE (1)
0x81, 0x02,          //     INPUT (Data,Var,Abs)
0x95, 0x01,          //     REPORT_COUNT (1)
0x75, 0x05,          //     REPORT_SIZE (5)
0x81, 0x03,          //     INPUT (Cnst,Var,Abs)
0x05, 0x01,          //     USAGE_PAGE (Generic Desktop)
0x09, 0x30,          //     USAGE (X)
0x09, 0x31,          //     USAGE (Y)
0x15, 0x81,          //     LOGICAL_MINIMUM (-127)
0x25, 0x7f,          //     LOGICAL_MAXIMUM (127)
0x75, 0x08,          //     REPORT_SIZE (8)
0x95, 0x02,          //     REPORT_COUNT (2)
0x81, 0x06,          //     INPUT (Data,Var,Rel)
0xc0,                //   END_COLLECTION
0xc0                 // END_COLLECTION
```

USB HID - Probing


```
static int usbhid_parse(struct hid_device *hid) {
        [...]

        if (!rsize || rsize > HID_MAX_DESCRIPTOR_SIZE) {
                dbg_hid("weird size of report descriptor (%u)\n", rsize);
                return -EINVAL;
        }

        if (!(rdesc = kmalloc(rsize, GFP_KERNEL))) { .. }

        ret = hid_get_class_descriptor(dev, interface->desc.bInterfaceNumber, HID_DT_REPORT, rdesc, rsize);

        if (ret < 0) { .. }
```

USB HID - Probing

```
int hid_open_report(struct hid_device *device) {
        [..]

        start = device->dev_rdesc;
        size = device->dev_rsize;

        buf = kmemdup(start, size, GFP_KERNEL);
        [..]
        start = kmemdup(start, size, GFP_KERNEL);

        kfree(buf);

        device->rdesc = start;
        device->rsize = size;

        parser = vzalloc(sizeof(struct hid_parser));
```

We get the report descriptor and make 2 copies of it!

→ Arbitrary write

Exploitation in the dark

Development environment

Exploitation Approach

Even if the bug is there...

- How do we develop an exploit without any means to debug?
  - No crash logs, no serial port, nothing
- How do we make sure we can predictably win the race?
- What is our payload supposed to look like?

Aaand the winner is: QEMU

Development environment: QEMU

Exploitation Approach

- Spend a lot of time making an environment close to the real device
  - Benefit: Full control & ability to debug the attack
    - Critical to build a deep understanding of all the steps
- Challenges:
  - An obvious consequence is that the target binary will be different
    - At least ensure all critical code paths & data structures are identical!
  - Also the device's activity level may vary
    - More activity == more chance of losing the race

  - **How accurate is the emulation?**

Development environment: QEMU

Exploitation Approach

- How to test the attack?
  - We can use the QEMU emulated devices
    - usb-mouse → MIDI device (with invalid configuration)
    - usb-tablet → Delivers payload through HID report descriptor

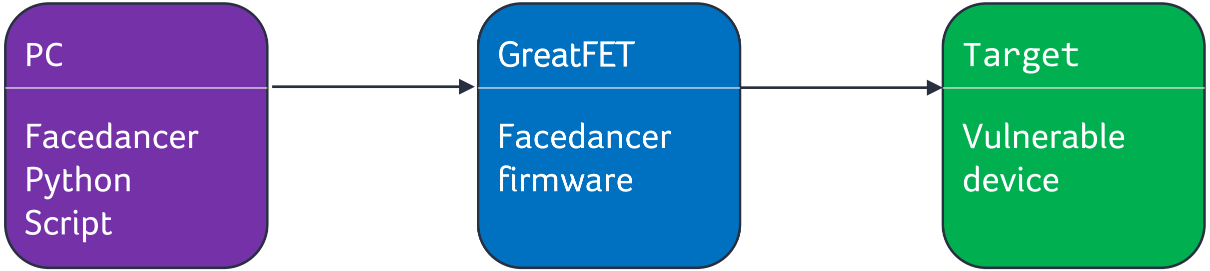
Major surprise:

When we did it on the real device, it almost immediately worked!

Payload delivery method

Exploitation Approach

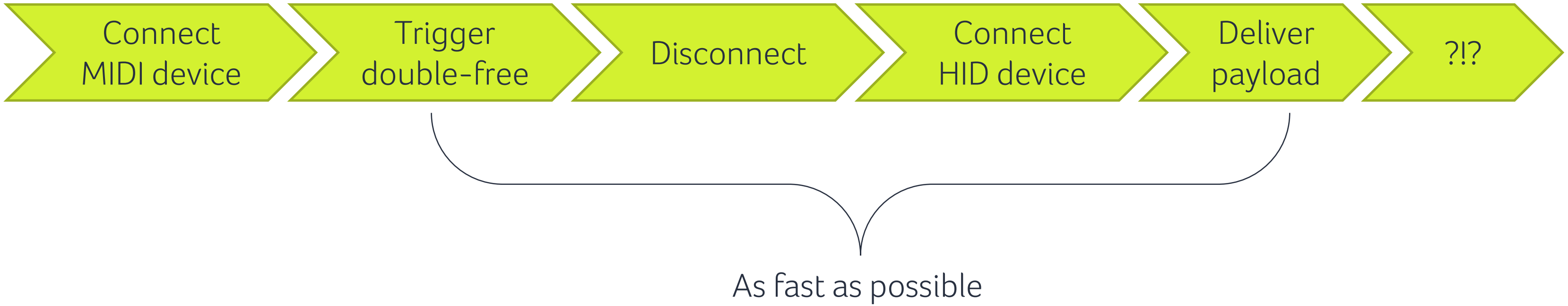
- No off-the-shelf device will:
  - Cause the double-free using a MIDI device
  - Allow us to deliver the payload in an HID report descriptor



Payload delivery method

Exploitation Approach

- No off-the-shelf device will:
  - Cause the double-free using a MIDI device
  - Allow us to deliver the payload in an HID report descriptor



Exploit payload

Exploitation Approach

Code writeable?

Yes

Overwrite kernel code → Arbitrary code execution

No

Data only attack

Where to hijack the code?

Exploitation Approach

```
.text:FFFF00000886F1B0 device = X22                          ; hid_device *
.text:FFFF00000886F1B0                 BL              _mcount
.text:FFFF00000886F1B4                 LDR             W0, [device,#0x1BC0]
.text:FFFF00000886F1B8                 TBNZ            W0, #1, loc_FFFF00000886F3C0
.text:FFFF00000886F1BC                 LDR             X0, [device] ; src
.text:FFFF00000886F1C0                 CBZ             X0, loc_FFFF00000886F3E8
.text:FFFF00000886F1C4                 LDR             W1, [device,#8]
.text:FFFF00000886F1C8                 MOV             W2, #0x24000C0 ; gfp
.text:FFFF00000886F1D0                 STR             W1, [X29,#0x70+len]
.text:FFFF00000886F1D4                 MOV             W1, W1  ; len
.text:FFFF00000886F1D8                 BL              kmemdup
.text:FFFF00000886F1DC                 MOV             X20, X0
.text:FFFF00000886F1E0                 CBZ             X0, loc_FFFF00000886F428
.text:FFFF00000886F1E4                 LDR             X0, [device,#0x1BB0]
.text:FFFF00000886F1E8                 LDR             X3, [X0,#0x60]
.text:FFFF00000886F1EC                 CBZ             X3, loc_FFFF00000886F308
.text:FFFF00000886F1F0                 ADD             X2, X29, #0x5C ; '\'
.text:FFFF00000886F1F4                 MOV             X1, X20
.text:FFFF00000886F1F8                 MOV             X0, device
.text:FFFF00000886F1FC                 BLR             X3
.text:FFFF00000886F200
.text:FFFF00000886F200 loc_FFFF00000886F200                  ; CODE XREF: hid_open_report+17C↓j
.text:FFFF00000886F200                 LDR             W1, [X29,#0x70+len] ; len
.text:FFFF00000886F204                 MOV             W2, #0x24000C0 ; gfp
.text:FFFF00000886F20C                 BL              kmemdup
.text:FFFF00000886F210                 MOV             X19, X0
.text:FFFF00000886F214                 MOV             X0, X20 ; x
.text:FFFF00000886F218                 BL              kfree
.text:FFFF00000886F21C                 CBZ             X19, loc_FFFF00000886F428
.text:FFFF00000886F220                 LDR             W1, [X29,#0x70+len]
.text:FFFF00000886F224                 MOV             X0, #0x18100 ; size
.text:FFFF00000886F22C                 STR             X19, [device,#0x10]
```

Payload design

Exploitation Approach

- Started with testing different payloads in QEMU
  - First payload only called called printk() & crashed
  - Next version called usb_get_string() (observable from the outside)
  - Final version could run an indefinite amount of times!

Exploit payload

Exploitation Approach

Max 512b

Destination pointer

- Destination address of the payload (kmalloc → kmemdup → **kmemdup**)

Cleanup

- Set freelist ptr to 0
- Set hid->collection & hid->rdesc to 0

Payload

- Disable protections (e.g. SELinux)
- Extract data (e.g. cryptographic keys)
- Run shell commands

Exploit payload

Exploitation Approach

Max 512b

Destination pointer	• Destination address of the payload (kmalloc → kmemdup → **kmemdup**)
Restore code	• Copy back part of the original code we overwrote with our payload so we can run the attack again
Cleanup	• Set freelist ptr to 0 • Set hid->collection & hid->rdesc to 0
Payload	• Disable protections (e.g. SELinux) • Extract data (e.g. cryptographic keys) • Run shell commands
Return	• Restore callee saved registers • Return an error code to stop the probing process

Exploit payload

Exploitation Approach

Destination pointer
- Destination address of the payload (kmalloc → kmemdup → **kmemdup**)

Restore code
- Copy back part of the original code we overwrote with our payload so we can run the attack again

Cleanup
- Set freelist ptr to 0
- Set hid->collection & hid->rdesc to 0

Payload
- Disable protections (e.g. SELinux)
- Extract data (e.g. cryptographic keys)
- Run shell commands

Return
- Restore callee saved registers
- Return an error code to stop the probing process

Run shell commands

Exploitation Approach

- Standard method to run arbitrary commands is using `run_cmd()`
  - However: cannot run from interrupt context
    - `usb_hub_wq` runs under this context
- Alternative:
  - Use `system_wq` to schedule start of new process
- `orderly_reboot()` / `orderly_shutdown()`
  - Both have writeable commands they are going to execute → overwrite & call

Run shell commands

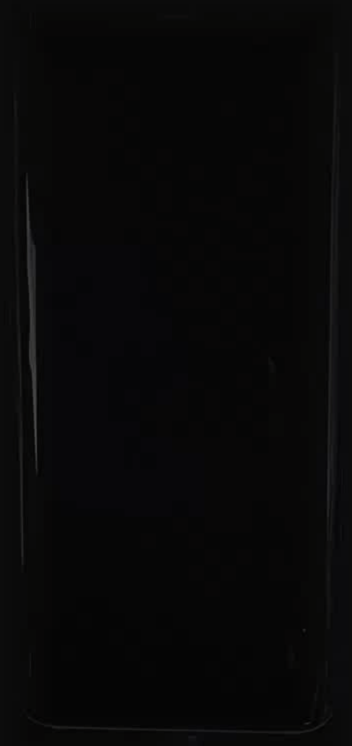
Exploitation Approach

- What shall we run?
  - Existing binary in rootfs (e.g. wget, nc, ...)
  - When a USB stick is auto-mounted → Run binary from USB stick (e.g. reverse shell)
  - Otherwise?
    - Make device node for USB stick
    - Mount USB stick
    - Then run the binary
- What about SELinux preventing to spawn processes?
  - Disable it! (selinux_enforcing = 0, patching the policydb, unhooking the LSM hooks)

Demo time!

```
$
```

```
Listening on [0.0.0.0] (family 2, port 1337)
```



NOTE: This device is modified for the purpose of this demonstration and is NOT vulnerable when the latest firmware version is installed.

Discussion

Attack challenges

- Winning the race
  - Midi and HID probe need to happen before kernel panics due to corruption
  - Midi and HID probe might happen on different cores due to scheduling
    - High chance (~50-80%) of winning the race when system is idle
- Cache behavior
- Exploit mitigations

Exploit mitigations

- ASLR?
  - Leak kernel pointer (e.g. uninitialized memory vulnerability)
  - Alternatively brute-force destination address (only when crashing the device several times is acceptable)
- Write protected kernel code / rodata
  - Data-only attack / targeting code of priv. umode process through physmap
  - Might be hard to not crash the kernel due to heap corruption!
  - Also: demo device discards writes but caches them until write-back
    - Still exploitable!
- Freelist hardening / randomization
  - Might be truly making this attack hard / infeasible

Applicability

- What about similar bugs?
  - Likely exploitable using this approach when it results in a double free on a low-activity slab
- Exploitation steps may differ
  - Architecture specific characteristics (cache, etc.)
  - Different Linux kernel configurations
  - Implemented exploit mitigations
  - Vendor specific customizations

Exploitation Requirements

[ ] Linux based device vulnerable to CVE-2016-2384

[ ] Physical access

[-] Unprivileged code execution

[-] Networking interface

[+] Low activity slab

Takeaways

- Fixed vulnerabilities upstream can take a long time to propagate
- Invest in building debug capabilities for your target!
  - Either through emulation or other means
- USB attacks are powerful & physical access might be all you need
  - In some cases it's your only attack vector!