USBFuzz: A Framework for Fuzzing USB Drivers by Device Emulation

Hui Peng\textsuperscript{1}, Mathias Payer\textsuperscript{2}
Testing drivers is challenging

- Drivers depend on hardware/devices
- Hard to generate unexpected device inputs
- Many recent attacks/CVEs
Motivation

● Defenses against peripheral attacks are limited
  ○ Rule-based authorization policy (USBGuard) and USB Firewalls (LBM, USBFilter)
    ■ Detect only known bugs
  ○ Isolation based approaches (Cinch)
    ■ Too expensive, not used in practice

● Fuzzing is a widely used automatic software testing technique

● We propose a framework to apply fuzzing to **USB drivers**
  ○ Fixing bugs is better than defending against their exploitation
Threat Model

- Device controlled by attacker via:
  - Prepared device through physical interfaces, or
  - Hijacking networked interfaces (e.g., USBRedir, etc)

- Attack vector
  - Focusing on unexpected data from device side
USB Fuzzing: Challenge & Existing Approaches

● Challenge
  ○ *How to feed random device input to drivers*

● Existing Approaches
  ○ Dedicated Hardware
    ■ Umap + FaceDancer
    ■ Hardware cost, not scalable, etc
  ○ Data injection in the IO stack
    ■ Syzkaller usb-fuzz, PeriScope
    ■ Not portable, some code paths are missed
  ○ Data injection through networked USB interface
    ■ vUSBf
    ■ Only supports dumb fuzzing
USBFuzz: Device Emulation

- Using emulated USB device in a virtualized kernel
  - Feeding random device inputs to USB drivers
  - A host memory area exported to the guest system
    - Can be used for coverage collection

- Advantages
  - Cheap, scalable, portable
  - Supports coverage guided fuzzing
USBFuzz Design

- **Guest system**
  - Runs target system
  - Fuzzing device
  - Communicating device
  - User Mode Agent

- **Fuzzer**
  - Runs in host system
  - Controls test execution
  - Collect coverage info
Test Execution

- A test starts from attaching the fuzzing device to guest system
- Drivers are tested while performing IO ops on the fuzzing device
- User Mode Agent detects end of a test by scanning kernel logs
Evaluation

- Coverage guided fuzzing on Linux kernel
  - Adapted KCOV to collect coverage info
- Dumb fuzzing on FreeBSD, Windows and MacOS
  - Seeded by inputs generated when fuzzing Linux kernel
Bugs and CVEs

- **Bugs**

<table>
<thead>
<tr>
<th>OS/Kernel</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>26 new bugs, 16 of them are memory bugs</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>One in a USB bluetooth dongle driver</td>
</tr>
<tr>
<td>Windows</td>
<td>4, resulting Bluescreen on Windows 8 and Windows 10</td>
</tr>
<tr>
<td>MacOS</td>
<td>3, two resulting unplanned restart and one resulting freeze</td>
</tr>
</tbody>
</table>

- **CVEs**

Fuzzing throughput

- 0.1 ~ 3.0 exec/sec
- Most of the execution time is spent on OS-device interaction
- Similar throughput observed in syzkaller usb-fuzz

A sample of execution speed of USBFuzz

A sample of execution speed of syzkaller usb-fuzz
Case Study (1)

- USB devices are described by **Device Descriptors**
- USB standard defines the first 2 bytes
  - The first byte indicates the length
  - The second byte indicates the type
- Descriptors are read from the device side

```c
struct usb_descriptor_header {
    __u8 bLength;
    __u8 bDescriptorType;
} __attribute__((packed));
```

Data structure representing the first 2 bytes of a descriptor

```c
struct usb_otg_descriptor {
    __u8 bLength;
    __u8 bDescriptorType;
    __u8 bmAttributes;
} __attribute__((packed));
```

Descriptor for USB On-The-Go
Case Study (2)

- Bug detected in case of malicious descriptors
- Attackers can masquerade long descriptors using short ones

```c
int __usb_get_extra_descriptor(char *buffer, unsigned size, unsigned char type, void **ptr) {
    struct usb_descriptor_header *header;
    while (size >= sizeof(struct usb_descriptor_header)) {
        header = (struct usb_descriptor_header *)buffer;
        if (header->bLength < 2) { return -1; }
        if (header->bDescriptorType == type) {
            *ptr = header;
            return 0;
        }
        buffer += header->bLength;
        size -= header->bLength;
    }
    return -1;
}
```
Case Study (3)

- Security Impact
  - Allowing Out-Of-Bounds Access

```c
static int usbEnumerateDeviceOTG(struct usb_device * udev) {
    // ......
    struct usb_otg_descriptor * desc = NULL;
    err = __usb_get_extra_descriptor(udev->rawdescriptors[0],
        le16_to_cpu(udev->config[0].desc.wTotalLength), USB_DT_OTG, (void **) &desc);
    if (err || !(desc->bmAttributes & USB_OTG_HNP))
        return 0;
    // ......
}
```

```
struct usb_otg_descriptor {
    __u8 bLength;
    __u8 bDescriptorType;
    __u8 bmAttributes;
} __attribute__ ((packed));
```
Demo/Windows - Bluescreen of Death
Demo/MacOS - Unplanned Restart
Conclusion

● Testing drivers is challenging
● USBFuzz provides a device emulation based approach fuzz USB drivers
● USBFuzz is cheap, portable and flexible
● So far USBFuzz has found:
  ○ 26 bugs in Linux
  ○ one bug in FreeBSD
  ○ 4 bugs in Windows
  ○ 3 bugs in MacOS

https://github.com/HexHive/USBFuzz
EOF
Data injection in syzkaller usb-fuzz