POTUS: Probing Off-The-shelf USB drivers with Symbolic fault injection

James Patrick-Evans, Lorenzo Cavallaro, and Johannes Kinder
Royal Holloway, University of London

11th USENIX Workshop on Offensive Technologies – August 14-15, 2017
USB Device Drivers are a Problem

• Drivers are buggy
  • Drivers have 3-7 times higher number of bug reports than the core kernel

• Lots of old device drivers
  • OSs want to support as many devices as possible

• High privilege
  • Kernel modules run in ring 0

USB drivers present an easy attack vector for exploitation!
POTUS

• Semi-Automated testing framework for Linux USB device drivers
  • Fault Injection
  • Concurrency Fuzzing
  • Selective Symbolic Execution

• Emulate generic USB devices
  • Detect memory errors and data races in the kernel
What is S2E?

Select module of interest for symbolic execution

Select analysis plugins for evaluation or path selection

/usr/bin/cp
uas.ko
scsi.ko
uas.ko
/ usr/bin/cp
USB 101

• Master-Slave protocol, controlled by a Host Controller Interface (HCI)

• At the “software level” – communication is done via Universal Request Blocks (URBs)

• Transfer data to endpoints by scheduling URB transfers

• Devices identify by providing a set of descriptors
System Overview

S2E
- Plugins
  - Code Selector
  - State Tagging
  - Path Pruning
  - Bug Detector

QEMU
- usb-generic
- UHCI
- USB redir

Linux VM
- SystemTap
- USB Driver
- Driver Exerciser
- KMemleak
- KASAN
- KTSAN

Real USB device
usb-generic

- Extended QEMU device model
- USB from JSON
  - String Descriptors
  - Device Descriptors
  - Interface Descriptors
  - Class Descriptors
  - Endpoint Descriptors

```json
{
  bInterfaceNumber: 0,
  bNumEndpoints: 2,
  bInterfaceClass: 8,
  bInterfaceSubClass: 6,
  bInterfaceProtocol: 50,
  eps: [
    {
      bEndPointAddress: 1,
      bmAttributes: 1,
      wMaxPacketSize: 512
    },
    ...
  ]
}
```

```
$ qemu-system-x86_64 -device usb-generic,cfg=config.json
```
• Host defined time, size and direction of URBs

• Inject symbolic or concrete data into IN requests, disregard data from OUT requests

• Inject delays and errors into URB requests by returning STALL, NYET and NAK

• Predefined routines for standard requests used during device enumeration e.g. GET_DESCRIPTOR
Exercising Drivers

- Random **stress test** of interfaces exposed by drivers
- Expose concurrency errors by running operations simultaneously
- Pick an operation at random from a weighted tree which is performed on file descriptors
- Every time `sys_open` is called, **fork** the user space driver exerciser, operate on both file descriptors
Injecting Faults

- **SystemTap** and kprobes kernel infrastructure
- Inject symbolic data and **symbolic faults**
- Typically, errors in the kernel are of the form – **ERRNO**
- Tapset libraries for hooking core kernel modules

```c
probe module("v4l2").function("video_register_device").return {
    child = s2e_fork()
    if (child) {
        s2e_log(__FUNC_NAME__ . " :: Injecting fault.\n")
        s2e_annotate(@FAULT_KEY, annotation + 1)
        video_unregister_device(@entry($vdev))
        $return = s2e_get_symb_fault(32)
    }
}
```
Airspy SDR

• Local DoS

• Present since 2013 or Linux 3.17 – 4.6

• drivers/media/usb/airspy/airspy.c

• CVE-2016-5400

• Debian/Ubuntu/Arch Linux
```c
static int airspy_probe(struct usb_interface *intf, const struct usb_device_id *id)
{
    struct airspy *s = kzalloc(sizeof(struct airspy), GFP_KERNEL);
    ret = v4l2_device_register(&intf->dev, &s->v4l2_dev);

    /* Register controls */
    v4l2_ctrl_handler_init(&s->hdl, 5);
    s->lna_gain_auto = v4l2_ctrl_new_std(&s->hdl, 
        V4L2_CID_RF_TUNER_LNA_GAIN_AUTO, 0, 1, 1, 0);
    ...

    ret = video_register_device(&s->vdev, VFL_TYPE_SDR, -1);
    if (ret) {
        dev_err(s->dev, "Failed to register as video device (%d)\n", ret);
        goto err_unregister_v4l2_dev;
    }
    ...

    return 0;
}

err_free_controls:
    v4l2_ctrl_handler_free(&s->hdl);
err_unregister_v4l2_dev:
    v4l2_device_unregister(&s->v4l2_dev);
err_free_mem:
    kfree(s);
    return ret;
}
```

Can be triggered purely from hardware!
Lego USB Tower

• Local DoS or Local Privilege Escalation

• Present since 2002 or Linux 2.6 – 4.7

• drivers/usb/misc/legousbtower.c

• CVE-2017-XXXX

• Debian/Ubuntu/Arch/Fedora/RHEL
static int tower_probe (struct usb_interface *interface, const struct usb_device_id *id){
    ...
    /* we can register the device now, as it is ready */
    retval = usb_set_intfdata (interface, dev);
    retval = usb_register_dev (interface, &tower_class);
    ...

    /* get the firmware version and log it */
    result = usb_control_msg (udev,
        usb_rcvctrlpipe(udev, 0),
        LEGO_USB_TOWER_REQUEST_GET_VERSION,
        USB_TYPE_VENDOR | USB_DIR_IN | USB_RECIP_DEVICE,
        0, 0, get_version_reply, sizeof(*get_version_reply),
        1000);
    if (result < 0) {
        dev_err(idev, "LEGO USB Tower get version control request failed\n");
        retval = result;
        goto error;
    }

    error:
    kfree(get_version_reply);
    tower_delete(dev);
    return retval;
}
static inline void tower_delete (struct lego_usb_tower *dev)
{
    tower_abort_transfers (dev);

    /* free data structures */
    usb_free_urb(dev->interrupt_in_urb);
    usb_free_urb(dev->interrupt_out_urb);
    kfree (dev->read_buffer);
    kfree (dev->interrupt_in_buffer);
    kfree (dev->interrupt_out_buffer);
    kfree (dev);
}

static ssize_t tower_write (struct file *file, const char __user *buffer,
                     size_t count, loff_t *ppos)
{
    ...
    bytes_to_write = min_t(int, count, write_buffer_size);
    dev_dbg(&dev->udev->dev, "%s: count = %zd, bytes_to_write = %zd\n",
            __func__, count, bytes_to_write);

    if (copy_from_user (dev->interrupt_out_buffer, buffer, bytes_to_write)) {
        retval = -EFAULT;
        goto unlock_exit;
    }
    ...
Exploit – Lego USB Tower

• Can we build a UaF that is a **data-only** attack and will bypass **SMEP**, **SMAP** and PAX’s **RAP**?

1. Inject maximum delay into REQUEST_GET_VERSION
2. Have user space call `sys_open()`
3. Wait for the 1 second timeout
4. Remap the `dev struct` in the general kernel cache
5. Change the values for `dev->interrupt_out_buffer`
6. Overwrite `{fs,e}{u,g}id`
7. Repeat 5-6 to overwrite **capabilities**
Future Work

• Slow!
  • Interval URBs – *usb_fill_int_urb(..., $interval)*
  • State explosion due to (symbex faults)driver operations

• Some paths we can’t explore!
  • Concretization of data – real device may help

• *usb-generic ⊆ USB 2.0*
  • USB On-The-Go (OTG), USB 3.0
Conclusion

• Abstract hardware interface
  • OS agnostic, CPU architecture agnostic

• Semi-Automated USB Driver Test Framework
  • Concurrency Fuzzing
  • Fault Injection
  • Selective Symbolic Execution

• Two 0-days in the Linux kernel, proof of concept for data-only LPE
Symbolic Execution

```c
int myfunc(int x)
{
    int r = 0;
    if(x > 8){
        r = x - 7;
    }
    if(x < 5){
        r = x - 2;
    }
    return r;
}
```