View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Royal Holloway - Pure

POTUS: Probing Off-The-Shelf USB Drivers with Symbolic Fault Injection

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

Abstract

USB client device drivers are a haven for software bugs, due to the sheer variety of devices and the tendency of maintenance to slip as devices age. At the same time, the high privilege level of drivers makes them a prime target for exploitation. We present the design and implementation of POTUS, a system for automatically finding vulnerabilities in USB device drivers for Linux, which is based on fault injection, concurrency fuzzing, and symbolic execution. Built on the S^2E framework, POTUS exercises the driver under test in a complete virtual machine. It includes a generic USB device that can impersonate arbitrary devices and implements a symbolic fault model. With our prototype implementation, we found and confirmed two previously undiscovered zero-days in the mainline Linux kernel. Furthermore, we show that one of these vulnerabilities can lead to a data-only exploit affecting even hardened systems protected with the latest software and hardware defenses.

1 Introduction

Device drivers are a critical part of any modern operating system (OS) due to their privileged access to hardware and the OS kernel. At the same time, drivers are challenging to maintain and keep bug free due to the number of devices requiring support. As a result, device drivers commonly contain many more bugs than other parts of the kernel: in a classic study, 70% of Linux device drivers were reported to have an error rate three to seven times higher than that of the core kernel [9].

Drivers for devices on the Universal Serial Bus (USB) have recently received particular attention in the vulnerability research community [28, 15, 26, 22, 13]. The plethora of USB devices and the widespread adoption of the USB standard makes them a high value target; in particular, a working exploit against a USB device driver can permit malware to spread to air-gapped devices.

Several mechanisms have been introduced to protect against kernel-level exploits, but as long as bugs persist in drivers, they pose a denial of service vulnerability at best and a stepping stone for a multi-step exploit at worst. A wide range of defense techniques are available today, including Kernel Address Space Layout Randomization (KASLR), Data Execution Prevention (DEP), Return Address Protection (RAP) [24], System Mode Execution Prevention (SMEP), System Mode Access Prevention (SMAP), and Control Flow Integrity [11]. All of these help raise the bar for developing a working exploit but do not prevent attacks entirely.

The main avenues for eliminating bugs from device drivers are testing and static analysis. Existing testing approaches for Linux mainly focus on the core USB module (*usbcore*) instead of individual drivers. This includes the USB test suite of the Linux Test Project [20] and hardware-based approaches such as FaceDancer [15], Teensy [4], and FrisbeeLite [12]. As any testing approach, these are incomplete, but the large number of possible interactions between device and driver poses a particular challenge. Coverity [3] is a mature static analyzer that regularly scans the Linux kernel. However, like other generic static analysis tools, it will miss bugs that require knowledge about the runtime environment and hardware, or that are based on complex and concurrent interactions between different components. In fact, even though the Linux kernel, including its USB device drivers, is regularly tested and statically analyzed, we were able to find critical new vulnerabilities with the approach presented in this paper.

We describe the concepts behind our tool POTUS (**p**robing **o**ff-**t**he-shelf **U**SB drivers with **s**ymbolic fault injection) that focuses on finding deep bugs requiring complex interaction with a matching hardware device. We base the design of our tool on an attacker model of a user with physical access to a USB port and guest user privileges; a typical real-world scenario would be an attacker plugging in a custom USB device while a user is logged in (necessary commands can be typed by sending keystrokes from the device masquerading as a keyboard).

We make the following contributions:

- We present a design for finding bugs in device drivers by combining fault injection with concurrency fuzzing and symbolic execution. We employ a search strategy prioritizing the exploration of relevant faults, and we confirm its effectiveness with two case studies where we found previously unknown (zero-day) vulnerabilities in drivers part of the standard Linux kernel.
- We introduce our method for simulating a virtual USB device that allows testing arbitrary client drivers. As it is built entirely as part of the QEMU virtual machine [2], it is both operating system and architecture agnostic. Our virtual device is capable of specifying most USB devices legal under the USB specification [27] and can intentionally deviate from it where necessary.

In the remainder of the paper, we provide technical background (§2), introduce the design of POTUS (§3), its implementation (§4), and present our findings about vulnerabilities in Linux USB drivers (§5). Finally, we discuss limitations (§6), contrast with related work (§7), and conclude (§8).

2 Background

We start by giving a brief introduction to selective symbolic execution (§2.1) and explaining some fundamental concepts of USB and its support in Linux (§2.2).

2.1 Selective Symbolic Execution (S^2E)

Symbolic execution is based on the idea of interpreting a program on *symbolic* instead of concrete data and having instructions manipulate symbolic expressions instead of concrete values. This allows to express the values of variables in terms of symbolic inputs at any point during execution of the program. Conditional instructions are interpreted by computing the symbolic expression for the condition and forking the program state if both outcomes are feasible, recording the outcome in the *path condition*. Dynamic symbolic (also called concolic) execution extends this idea by also executing the program concretely at the same time. Both suffer from *path explosion*, the generally exponential growth in the number of paths in a program.

Selective symbolic execution, a technique introduced in S^2E [8], reduces path explosion by switching between concrete and symbolic execution modes at module boundaries. S^2E uses QEMU [2] and KLEE [5] to provide a symbolic virtual machine that allows to inject symbolic data into arbitrary memory addresses. In S^2E, each code block (translation blocks from Jams Tiny Code Generator) that manipulates symbolic data is compiled to LLVM instructions and symbolically executed by the KLEE symbolic execution engine. A plugin system allows for interactions with the virtual machine and provides an interface to processes running in the guest.

2.2 USB

The USB specification supports a vast amount of features, such as hotplugging, generic class drivers, multi-master USB On-The-Go, and data and power transfer modes. The protocol follows a master-slave design in which a Host Controller Interface (HCI) communicates with USB client devices—'gadgets' in the USB specification terminology. USB HCIs implement the low-level protocols of timings, packet scheduling, and signaling for each version of the specification. This is abstracted to Universal Request Blocks (URBs), which form the logical basis of communication with gadgets. URBs contain a *device address* to signify the device connected on the bus and an *endpoint* to specify a channel to a device. They can be one of four types:

- **Control Transfers**: device configuration and signal control information;
- **Bulk Transfers**: large quantities of time-insensitive data;
- **Interrupt Transfers**: small quantities of time-sensitive data;
- **Isochronous Transfers**: real-time data at predictable bit-rates.

An endpoint provides an address and direction for URB transfers from the perspective of the host. Control transfers to endpoint 0 are special; all USB devices must implement bi-directional (IN and OUT) communication to this endpoint and it is used during device initialization.

USB in Linux The Linux kernel's USB subsystem has a modular design allowing new client device drivers to easily interface with hardware [10]. The Linux kernel maintains a kernel daemon thread called *khubd*, which is responsible for monitoring any USB hubs by communicating with the onboard HCI and configuring USB devices. When a USB device is plugged into a machine, the kernel enumerates the device's capabilities and configurations in a process called *USB enumeration*. In this process, *khubd* is awoken by the main kernel thread upon a new USB event being triggered via a Host Controller's hardware interrupts. The purpose of USB enumeration is to iterate over its configuration descriptors to determine power output, associated endpoint addresses, transfer types, speed, device class, etc.

Once enumeration is complete, the associated client driver's probe function is invoked with a reference to the USB device and a USB client device driver can then communicate with the device through library functions in *usbcore*. Such communication requests are sent as URBs to a HCI and forwarded on to the gadget.

Most Linux device drivers allow userspace processes to access hardware through special files, e.g., a character device, block device, socket, or named pipe. Typically this involves system calls on files located in *devfs* or *sysfs*, which in turn call functions in kernel drivers. For example, the driver for USB Mass Storage Devices wraps a SCSI device around a USB device; file operations to its node under `/dev/sd[a-zA-Z]{1-3}` trigger URB transfers to the corresponding USB device.

3 Design

We now introduce the design of POTUS. We start by explaining the underlying attacker model (§3.1) and then give a high-level overview of its components (§3.2).

3.1 Attacker Model

Our approach implements an attacker model where the adversary has a user account and physical access to USB ports. A typical scenario would be a workplace environment in which an employee has access to a machine and wants to achieve privilege escalation; or a visitor using a brief moment to plug in a USB device that implements both a human interface device to send keystrokes and a device targeting an exploitable driver.

Following this intuition, our attacker model has several degrees of freedom: first, it incorporates any possible interaction between devices and the operating system, which allows to trigger the driver loading, device enumeration, and driver-device setup stages. This dimension has been the focus of related work in the area [19, 8, 26]. Second, our attacker model includes system calls from usermode, i.e., the capability to interact with files created by device drivers. In particular, our attacker, as implemented in the driver exerciser, has the capability of (concurrently) performing system calls on file descriptors that trigger execution paths within the device driver. Third, our attacker model includes the ability to influence scheduling and cause memory allocation failures, which an attacker can achieve in practice by placing the CPU under load or exhausting memory.

3.2 System Overview

An overview of POTUS's architecture is shown in Figure 1. POTUS builds on S^{2}E (which in turn builds on QEMU) to run a full guest OS in a virtual machine.

The *USB driver* under test resides in the VM and receives inputs from the *driver exerciser* and one or several USB devices. The USB devices connect through QEMU's Universal Host Controller Interface (UHCI) and may be either virtual usb-generic devices or real devices passed through QEMU's USB redirector. For our case studies we relied on virtual devices only, but real devices may help to explore deep paths in involved protocols without modeling overhead. The driver exerciser runs inside the guest and enumerates—in a form of fuzz

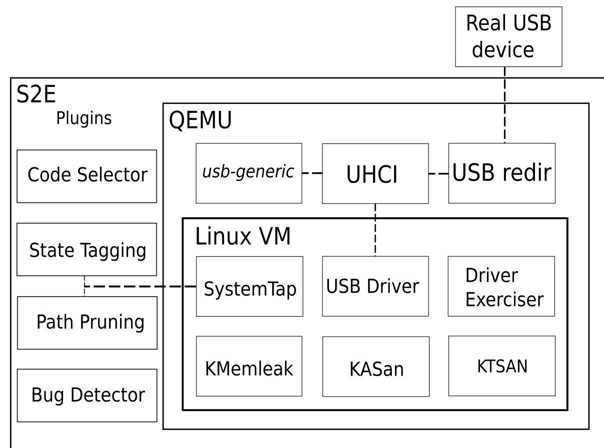

Figure 1: System overview of POTUS.

testing—possible interactions of a user process with the USB device.

We mark data sent from the device as symbolic to allow exploring all possible responses in S^{2}E, and we inject faults in the form of symbolic error codes. We use SystemTap scripts to interface between guest code and S^{2}E and inject data at function call or return sites. We implemented S^{2}E plugins for state tagging and path pruning to limit path explosion and prioritize states exploring different types of and locations of faults.

We rely on existing instrumentation tools to detect kernel-level bugs, namely Kernel Address Sanitizer (KASAN) for memory errors, Kmemleak for memory leaks, and Kernel Thread Sanitizer for data races. While the combination of these tools adds significant performance overhead, it allows us to have high confidence that we can detect any security-relevant errors as they occur. We also built a bug detector plugin for S^{2}E that intercepts kernel bug functions such as `BUG_ON()` and `panic()` calls and retrieves the kernel log file, the driver exerciser log, and various other system log files to help debug and interpret the results.

4 Implementation

We now describe the implementation of POTUS, in particular the generic virtual USB device (§4.1), the mechanism for fault injection (§4.2), the driver exerciser (§4.3), and its search strategy (§4.4).

4.1 The usb-generic Device

To exercise a USB device driver under all possible configurations, we created the virtual usb-generic device. It implements a generic USB device with configurable device descriptors and a symbolic model for data transfer.

The USB specification contains five descriptors: string descriptors, device descriptors, interface descriptors, configuration descriptors, and endpoint descriptors. usb-

generic can be configured to impersonate any possible USB device by configuring its descriptors in a set of configuration files. In principle, this data could also be extracted from the target driver automatically. Once the descriptors are configured, the usb-generic device is ready to be used for testing any driver; further customization is not required, but possible.

There are four URB types used for communication between a USB gadget and the host controller interface (see §2.2). By default, usb-generic ignores the content of OUT URBs and responds to IN URBs by writing the requested amount of data and marking it as symbolic for S^2E. We found that in practice the combination of symbolic data and a host-specified content length is sufficient to thoroughly exercise our target drivers, given enough time. However, usb-generic also provides the option for further (compile-time) customization by installing driver-specific callbacks to respond differently to each URB type. We can further separate functionality by device, interface, class, and endpoint request codes within per Device ID and Vendor ID segments. Finally, the usb-generic device can also be instructed to respond negatively to individual URBs to influence packet scheduling, inject delays, or intentionally violate the USB specification. Outside of vulnerability discovery, we believe that usb-generic can also be used by developers to aid writing device drivers in absence of a physical device, in particular using fine-granular callbacks.

4.2 Fault Injection

POTUS injects faults into the running kernel using SystemTap [25]. SystemTap allows to place *probes* into arbitrary kernel locations to gather information or inject compiled C code to modify kernel data structures. POTUS contains SystemTap modules for automatically injecting faults into core kernel submodules used by USB client device drivers such as *usbcore* and memory allocation mechanisms, an example of which can be seen in Listing 1.

In particular, this example shows how fault injection interacts with symbolic execution: the function will fork the current state (as long as the per-path fault limit is not exceeded (see §4.4) and return success in one state and a symbolic fault code in the other. The symbolic fault code allows to explore all possible error codes at once, which would be impossible with purely concrete fault injection. The Linux kernel uses negative `errno` return codes to indicate errors; we therefore constrain the symbolic expressions for the return value on error paths to be negative. In addition to returning symbolic error codes, our fault model also injects the maximum admissible *delay*. While this is by no means a complete approach to verifying concurrent code, it is often just enough "fuzz" to expose concurrency bugs.

```
probe module("usbcore")
  .function("usb_bulk_msg").return {
  nfaults = s2e_get_annotation(@FAULT_KEY)
  if (nfaults < @FAULT_LIMIT) {
    child = s2e_fork_state(__FUNC_NAME__ .
        " fork")
    if(child) {
      s2e_annotate(@FAULT_KEY, nfaults+1)
      if (@SYMBOLIC_FAULTS)
        $return = s2e_get_symb_fault(32)
      else
        $return = -1
      next
    }
  }
  ...
}
```

Listing 1: SystemTap probe for injecting faults into all `usb_bulk_msg` functions.

Some device drivers register asynchronous callbacks from the resulting URB inline; in that case we must manually write a short SystemTap probe for each. Our library of fault injectors works on a per module basis, i.e., it only injects faults into kernel threads that have a call stack associated with the target module under test.

4.3 Driver Exerciser

The driver exerciser in POTUS initiates operations on the device from userspace. It randomly invokes file operations on the target device driver's file descriptors; to expose concurrency bugs, it can initiate multiple concurrent operations on the same or different file descriptors. We implemented this by using symbolic execution to search depth first through a weighted tree of operations (see Table 1), initially instantiating an active file descriptor with `open()`. Currently we support device drivers exposing a socket, character device or block device file, which covers the vast majority of USB device drivers.

Figure 2 shows an example execution path for a guest user opening a character device file owned by the *legousbtower* driver. The user initiates the interaction by calling `sys_open` on a file whose file operations for `open()` map to a callback within the *legousbtower* driver. Should the driver attempt to allocate memory or call other kernel subsystems for which fault injection is currently active, POTUS forks the system state and returns a symbolic fault in one state, indicated by the red path. In the fault-free path, the driver callback calls `usb_submit_urb` with a device request of a custom `TOWER_RESET` device reset code, which is passed to the HCI via `usb_hcd_submit_urb`. At this point, the system state is forked again, with the fault-free path returning successfully, and the other path failing the URB transfer after the maximum delay.

Function	Description
open()	Open a new file descriptor corresponding to one of the exposed device files. Upon being called, the guest driver exerciser forks into two processes, with both accessing the device concurrently to try and trigger concurrency bugs.
close()	Close the active file descriptor.
connect()	Simulate a physical hardware connection.
disconnect()	Simulate a physical hardware disconnection.
read()	Perform a sys_read on the currently active file descriptor of random length. Discard the data read.
write()	Perform sys_write on the currently active file descriptor of random length. Data written is made symbolic.
poll()	Perform sys_poll on the currently active file descriptor. Request all events for a randomized timeout.
lseek()	Perform sys_lseek on the currently active file descriptor. Seek to a random offset for the current active file descriptor.
ioctl()	Perform sys_ioctl on the currently active file descriptor. Currently implemented for driver specific ioctl calls and arguments however may be made for generic by providing a symbolic call code and argument.
send()	Perform sys_send on the currently active socket. Send a buffer of symbolic data with a random length and symbolic or concrete flags. Allows for sending data on sockets of domain AF_INET and type SOCK_STREAM or SOCK_DGRAM.
recv()	Perform sys_recv on the currently active socket. Receive a random sized buffer with symbolic or concrete flags and discard any data read. Allows for receiving data on sockets of domain AF_INET and type SOCK_STREAM or SOCK_DGRAM.

Table 1: File descriptor operations implemented in the driver exerciser.

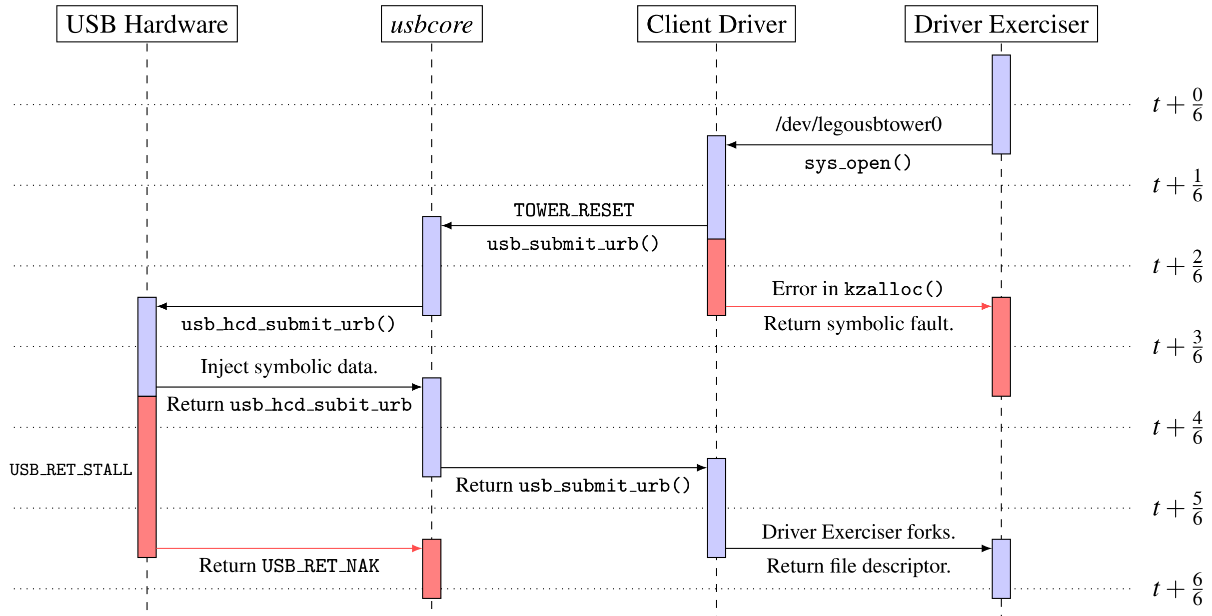

Figure 2: Example of flows for viable code paths from the driver exerciser through the Linux USB stack. The diagram shows fault injection and fork points for the *legousbtower* client driver.

Driver Tested	LOC	Driver-specific Probes	Functions in Driver	Exposed Interface
AirSpy	1,108	1	32	*v4l2* device
Apple USB Display	369	1	8	Backlight device
Chaos Key	579	0	11	Character file
Cytherm	407	0	13	*Sysfs* files
IO Warrior	919	2	15	Character file
Lego USB Tower	982	2	15	Character file

Table 2: List of device drivers tested.

4.4 Path Prioritization

Since we see POTUS as complementary to existing testing and bug finding approaches, we particularly focus our vulnerability search on deep bugs arising from concurrency errors, faults, and their interplay. Our intuition is that those bugs that are detectable by light-weight static analysis will have already been found in the Linux kernel. Still, the multitude of low-level concurrency primitives in a Linux kernel running on modern hardware harbor great potential for lingering concurrency bugs.

The combination of symbolic faults, concurrency fuzzing with delays, and symbolic data aggravates the state explosion problem in symbolic execution. Since we are interested in exploring deep paths, we prioritize a primary path without faults or delays, and use it to spawn new states at potential fault and delay injection points. POTUS attaches a map of annotations to each state, which is cloned upon forking; we use this map to track the number of faults already injected into the state (see Listing 1, line 3) and the number of children created. We use this to limit the number of fork points by preventing further fault injection when the limit of faults is reached. This balances general code coverage with the exploration of fault routines. The intuition behind this optimization is that paths with a high number of faults will likely lead to exploring the same code numerous times without exploring new error handlers. We thus prioritize complex driver-device exchanges over exploring the entire, generally infinite, set of possible interactions.

5 Evaluation

We now present a preliminary evaluation of our approach; we were interested in evaluating POTUS's ability to effectively find bugs and its potential for generalizing to different types of USB drivers. We first discuss our setup and methodology (§5.1), explain how POTUS applies to six target USB device drivers (§5.2), and then provide details and an assessment of exploitability for the two previously unknown vulnerabilities we discovered: CVE-2016-5400[1] resulted in a denial-of-service attack on Linux kernels 3.17–4.6 (§5.3); CVE-2017-XXXX[2] allows an arbitrary write/read primitive affecting Linux kernels pre 2.6–4.6 since 2003 (§5.4).

5.1 Experimental Setup

All of our experiments were run on a Ubuntu 12.04 LTS with dual Intel(R) Xeon(R) CPU E5-2640. We used our own fork of S^{2}E based on the latest version as of 2016-11-01 and Debian Sid as the guest OS running a custom vanilla Linux 4.6 kernel, which only enables the required modules and keeps a minimal guest OS. We dynamically loaded `usb-generic` devices through QEMU's monitor interface and executed S^{2}E on all the available CPU threads of the host platform.

We developed an automation framework to control experiments, including booting the OS, inserting SystemTap probes, and loading client device drivers. The guest OS was executed with a QEMU emulated Core 2 Duo CPU and 1 GB of RAM. Overall, we ran each driver for up to one hour, exploring in the order of hundreds of states per experiment. We used S^{2}E in concolic mode, exploring a state until termination before switching to a new state, to explore deeper code paths.

POTUS's memory requirements are kept manageable (considering it is based on full-VM symbolic execution) by our path pruning strategy. For instance, in testing the Lego USB Tower driver for one hour with 32 S^{2}E processes, POTUS forked 488 different VM states and used 6.6 GB of RAM. As POTUS runs S^{2}E in concolic mode, it executes the driver exerciser to a fixed number of operations before terminating the state, which improves POTUS overall memory footprint as all the states resources do not have to be saved simultaneously.

5.2 Adapting to Target Device Drivers

To evaluate our claim of a generic testing tool suite for USB device drivers, we consider the effort it takes to test new drivers. We selected six open source USB device drivers that are included in the mainline Linux kernel, touch on several of its different subsystems, and have a significant number of lines of code. Table 2 lists the device drivers that we tested along with their lines of code,

[1] `https://nvd.nist.gov/vuln/detail/CVE-2016-5400`

[2] We have requested a CVE but are yet to receive an assigned ID.

number of functions, the exposed interface, and the number of driver-specific probes we had to write.

We could test Chaos key and Cytherm entirely with POTUS's default libraries for injecting faults, because these drivers rely only on synchronous `usbcore` library functions such as `usb_bulk_msg` to transfer data. Although the driver exerciser must be instructed to point to the corresponding device files, there was no need for additional SystemTap probes. Conversely, drivers that also rely on asynchronous `usbcore` library functions expose specific callbacks that have to be instrumented, and required one to two driver-specific probes (see Table 2). It is feasible to automatically address such contexts (e.g., we can modify our SystemTap libraries with inline C code to dereference forward referencing functions), but the currently necessary manual effort is minimal.

5.3 Airspy (CVE-2016-5400)

CVE-2016-5400 represents a memory leak vulnerability in a USB device driver for communicating with an Airspy Software Defined Radio (SDR), located under `drivers/media/usb/airspy/airspy.c` in the kernel source tree. The memory leak can be triggered purely from hardware to perform a Denial of Service (DoS) attack, crashing the host by plugging in a specially-crafted USB device. The USB device driver interacts with the Video For Linux 2 (V4L2) subsystem and, as a result, requires the allocation of `v4l2` device structures and registration with the subsystem. The programming error that led to the memory leak was situated in the drivers `probe` function; a function that is called when a new device associated with the driver is plugged into the host. The relevant code snippet can be seen in Listing 2 and shows that if the `video_register_device` function fails, the driver fails to free any of the control variables registered with the `v4l2` subsystem. POTUS's automatic fault injection identified this memory leak.

Exploitability. The Airspy kernel module was installed by default in most Linux distributions, including, but not limited to Ubuntu, Debian, Arch Linux, and Trisquel; it loads whenever a USB device with the Airspy device descriptor is plugged in.

An attacker can make `video_register_device` fail with a specially-crafted hardware as the Linux kernel only supports a maximum of 64 minor numbers for `VFL_TYPE_SDR` type devices attached to a host at any given time. By creating a USB device that acts as a hub and attaches 65 of the same devices, we can trigger the memory leak vulnerability. The sequence of connection and disconnection operations on the 65th device consumes all the available RAM and effectively triggers a DoS attack. We successfully verified the feasibility of this attack under POTUS testing framework.

```
static int airspy_probe(struct
    usb_interface *intf,
        const struct usb_device_id *id)
{
    ...
    v4l2_ctrl_handler_init(&s->hdl, 5);
    ...
    ret = video_register_device(&s->vdev,
            VFL_TYPE_SDR, -1);
    if (ret) {
        dev_err(s->dev, "Failed to...");
        goto err_unregister_v4l2_dev;
    }
    dev_info(s->dev, "Registered as ...");
    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;
}
```

Listing 2: Airspy probe function.

5.4 Lego USB Tower (CVE-2017-XXXX)

CVE-2017-XXXX is a Use-After-Free vulnerability that has existed in the Linux kernel's Lego USB Tower driver since 2003 (`drivers/usb/misc/legousbtower.c`). The driver is quite pervasive: it is compiled and available with the majority of Linux distributions, including the latest server editions of Ubuntu 16.04 LTS and Fedora 25. The vulnerability is a race condition that leads to a NULL pointer dereference; if remapped to a user-controlled memory location, it can be abused to escalate privileges or execute arbitrary code.

Listing 3 shows the driver probe function and entry point into the program. The function registers a character device file `/dev/usb/legousbtower[0-9]+` and proceeds to submit a request for the device firmware version. If this URB request fails, the driver then calls `tower_delete`, which deletes the device structures associated with the driver without checking for any active connection. Registering the device file grants file operations from userspace, an action which could happen before the probe function terminates. Listing 4 details the `tower_write` function, which maps to the `sys_write` system call and checks that the device is still connected before copying data from userspace into a local kernel buffer pointed to by `dev->interrupt_out_buffer`. If `tower_delete` is called after the write function checks that the device is connected, it will delete the `dev` structure, setting its value to NULL and causing a NULL pointer dereference in `tower_write`.

Exploitability. An attacker can create a USB device that will hold open or drop the control message for the

```
static int
tower_probe(struct usb_interface ...)
{
...
  /* register the device now, as it is
      ready */
  usb_set_intfdata (interface, dev);
  retval = usb_register_dev (interface,
      ...);
...
  /* 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:
  tower_delete(dev); return retval;
}
```

Listing 3: Lego USB Tower probe function.

boards firmware, providing with the time necessary to exploit the race condition. As the kernel executes in the same address space as userspace, an unprivileged user may map the NULL page (or use alternative techniques to work around limitations) to control the location of the data being written to. As an attacker also controls the data, it is possible to write an arbitrary payload to arbitrary memory locations, thus overwriting the local user id for the process to gain root privileges.

If the NULL page is mappable through the `sysctl` setting `mmap_min_addr` or by using a user account with the Linux personality of `MAP_PAGE_ZERO`, an adversary can easily force the location and data on a buffer written inside the kernel. Other methods, such as those that execute a *setuid* binary to remap existing memory have previously been shown to circumvent this protection[3]. Linux kernels before 2009 have no protection against mapping the NULL page and are thus easily exploitable using a specially-crafted USB device and a low-privileged guest user account to trigger the race condition. Upon further inspection, the device file exposed by this driver is made globally readable and writable on most systems by *udev*; something which happens after the probe function finishes and closes the race condition. This significantly lowers the impact of the vulnerability

[3]http://blog.cr0.org/2009/06/bypassing-linux-null-pointer.html

```
static int write_buffer_size = 480;
...
static ssize_t
tower_write (struct file *file,
    const char __user *buffer, size_t
        count, ...)
{
    struct lego_usb_tower *dev;
    size_t bytes_to_write;
    ...
    /* verify that the device wasn't
        unplugged */
    if (dev->udev == NULL) {
        retval = -ENODEV;
        pr_err("No device or device\
                unplugged %d\n", retval);
        goto unlock_exit;
    }
    /* wait until previous transfer is
        finished */
    while (dev->interrupt_out_busy) {
        if (file->f_flags & O_NONBLOCK) {
            retval = -EAGAIN;
            goto unlock_exit;
        }
    }
    /* write the data into
        interrupt_out_buffer
       from userspace */
    bytes_to_write = min_t(int, count,
            write_buffer_size);
    if (copy_from_user
       (dev->interrupt_out_buffer,
               buffer, bytes_to_write))
    ...
}
```

Listing 4: Lego USB Tower `tower_write` function.

but it may be used in a multi-stage exploit or to escape containers where the user already has an *fsuid* of 0.

Bypassing SMEP, SMAP and RAP. To assess the exploitability of CVE-2017-XXXX on a modern, security-hardened kernel, we decided to build a proof of concept of a local privilege escalation exploit that would work on the latest kernel at the time of development: Linux 4.6 with PaX's grsecurity patches applied. The underlying idea is to reallocate the same memory used by the struct deletion described above to control the location of the output buffer.

To be able to remap the same memory location as the kernelspace `dev struct`, we abused the Linux kernel's SLUB memory allocator that re-provisions previous allocations of the same size. Invoking `sys_sendmsg` allowed us to force the kernel to allocate arbitrarily sized memory in the general kernel cache. Once we identified the size of the message to send, we created a USB device to insert a one second delay for the `tower_probe`'s device firmware URB request, which

enabled us to consistently remap the same memory freed from `tower_delete`. Unfortunately, `sys_sendmsg` does not allow us to control the first 40 bytes of memory allocated, due to it being reserved for the messages metadata. In our case, while the main data structures were outside this region, it overlapped with a device mutex, which would block execution of the exploit indefinitely.

We relaxed the condition for exploitation and created a kernel module that would first leak memory addresses to bypass KASLR and discover the running process' `task struct` to overwrite its credentials and increase privileges and, secondly, allow us to allocate memory of arbitrary size on a kernel. Under such assumptions, we were able to dereference a pointer to the current processes `credentials struct` and overwrite `{u,g}id`, `e{u,g}id`, `s{u,g}id`, `fs{u,g}id` and `capabilities`. Our final exploit bypasses SMAP through the kernels own use of `copy_from_user`, temporarily disabling it without performing any buffer overflow or control flow hijacking, thus remaining unaffected by both SMEP and RAP. We believe that our relaxed exploitation conditions are realistic [16] and do not affect the feasibility of a successful attack. For instance, the second condition can be addressed by adjusting the `bMaxPacketSize` of the device descriptor to load data into that memory location to read and write data to the device.

6 Limitations

QEMU's current UHCI implementation supports USB devices up to USB 1.1. Although many devices are backwards compatible and simply transfer data at lower speeds, they may use some features of newer USB specifications that we therefore cannot test. For instance, usb-generic currently does not support a multi-master device setup and hence does not support the USB On-The-Go extension.

The implementation of `usb_submit_urb` in *usbcore* contains an `interval` parameter which specifies a time period for periodically polling the device, indefinitely. If we are masking data input from the device as completely symbolic data, and if each URB will result in at least one state being created from injected faults, then this presents an infinite set of possible paths to explore and further increases path explosion.

Furthermore, the high runtime overhead of symbolic execution in S^2E increases the frequency of timer interrupts relative to the execution of other instruction. As a result, we had to slow down QEMU's internal clock by a factor of five to avoid exploring only device polling in drivers using short intervals.

7 Related Work

Our approach draws on a range of previous work, which we compare and contrast to in this section.

Symbolic execution has been widely used for vulnerability discovery (e.g., [6, 14, 7]). Most closely related are S^2E itself [8] and in particular its predecessor project DDT [19]. Both have been used to find bugs in userspace applications and device drivers. POTUS builds on S^2E and expands it with features specific to the problem domain of Linux USB drivers. In a way, POTUS is a sister project to DDT in that it allows testing USB drivers on the latest Linux versions similarly to how DDT tested PCI drivers for Windows XP. However, DDT used fully symbolic PCI devices that would be too generic to allow meaningful exploration of devices communicating via a USB host controller.

Tonder and Engelbrecht [28] describe a hardware based mutation fuzzing scheme for USB. Their approach builds on the Facedancer project [15] to mutate the interactions of existing USB devices with the host. A pure software approach is inherently easier to deploy (e.g., where no related device is available) and more flexible; e.g., the reported 300ms delay in control transfers would make it difficult to discover timing-sensitive race conditions such as CVE-2017-XXXX. Furthermore, a hardware-only approach is likely to only exercise a very limited portion of the USB subsystem, in particular the USB device enumeration in *usbcore*.

Jodeit et al.'s [18] combined a physical USB device with a mutational based fuzzer to test an Apple iPod on Windows and found multiple software bugs in Windows XP drivers. Schumilo et al. [26] presented the software USB fuzzer vUSBf, which relies on QEMU's usbredir server to redirect URB packets from host emulated devices into the guest operating system. vUSBf mainly focuses on fuzzing values in USB descriptors and USB HID drivers and provides no systematic way of exercising device drivers.

NCC's *umap2*[4] allows to fuzz USB device drivers by recording traces from emulated devices and then fuzzing replays of the traces. The project relies on gadgetfs or Facedancer and a Python program to describe each device. The project is currently able to emulate 13 device classes, each specified in hundreds of lines of Python code. In contrast, POTUS provides significantly more automation, requiring typically to only adapt a few SystemTap scripts and configure the virtual device.

Software-implemented fault injection (SWIFI) is a widely used technique for testing the robustness of software. Natella et al. [23] provide a recent survey of the area. A flexible framework for fault injection at the level of libraries was presented in LFI [21]. LFI automatically generates error models for libraries, which we aim to also achieve at kernel level for POTUS in future work. A comparative study of fault injection techniques by Jar-

[4] `https://github.com/nccgroup/umap2`

boui et al. [17] showed that internal software faults or faults caused by device drivers could not be easily emulated by injecting faults at the system call level only. This validates our design choice in POTUS to allow fault injection at any point in the kernel, and in particular at the level of the kernel subsystem APIs. The impact of device drivers on the Linux kernel is a known cause for concern. For example, Albinet et al. [1] characterized the kernel's robustness based on the impact of faulty device drivers.

8 Conclusion

We have presented POTUS, a new approach for testing USB device drivers that, while still a work in progress, has found long existing bugs that previous state-of-the-art tools failed to find. As our approach is built on top of Free & Libre Open Source Software, it is easily extendable, adaptable and can work together with existing open source projects to provide further functionality. We have found two critical vulnerabilities in the latest version of the Linux kernel and built proof of concept exploits to explore their severity. All our code and configuration data are available as open source.

Acknowledgments

James Patrick-Evans was supported by the EPSRC and the UK government as part of the Centre for Doctoral Training in Cyber Security at Royal Holloway, University of London (EP/K035584/1).

References

[1] ALBINET, A., ARLAT, J., AND FABRE, J. Characterization of the impact of faulty drivers on the robustness of the Linux kernel. In *Proc. Int. Conf. Dependable Systems and Networks (DSN)* (2004), pp. 867–876.

[2] BELLARD, F. QEMU, a fast and portable dynamic translator. In *Proc. USENIX Annual Technical Conference, FREENIX Track* (2005), pp. 41–46.

[3] BESSEY, A., BLOCK, K., CHELF, B., CHOU, A., FULTON, B., HALLEM, S., GROS, C.-H., KAMSKY, A., MCPEAK, S., AND ENGLER, D. R. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM 53*, 2 (2010), 66–75.

[4] BURSZTEIN, E. Teensy - Malicious USB. `https://github.com/ebursztein/malusb`, 2016.

[5] CADAR, C., DUNBAR, D., AND ENGLER, D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. Operating System Design and Implementation (OSDI)* (2008), pp. 209–224.

[6] CADAR, C., GANESH, V., PAWLOWSKI, P. M., DILL, D. L., AND ENGLER, D. R. EXE: automatically generating inputs of death. In *ACM Conf. Computer and Communications Security (CCS)* (2006), ACM, pp. 322–335.

[7] CHA, S. K., AVGERINOS, T., REBERT, A., AND BRUMLEY, D. Unleashing Mayhem on binary code. In *IEEE Symp. Security and Privacy (S&P)* (2012), IEEE Computer Society, pp. 380–394.

[8] CHIPOUNOV, V., KUZNETSOV, V., AND CANDEA, G. S2E: a platform for in-vivo multi-path analysis of software systems. In *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2011), pp. 265–278.

[9] CHOU, A., YANG, J., CHELF, B., HALLEM, S., AND ENGLER, D. An empirical study of operating system errors. In *Proc. ACM Symp. Operating System Principles (SOSP)* (2001), pp. 73–88.

[10] CORBET, J., RUBINI, A., AND KROAH-HARTMAN, G. *Linux Device Drivers*, 3rd ed. O'Reilly, 2005.

[11] CRISWELL, J., DAUTENHAHN, N., AND ADVE, V. S. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *IEEE Symp. Security and Privacy (S&P)* (2014), pp. 292–307.

[12] DAVIS, A. *Fuzzing USB Devices using FrisbeeLite*. NCC Secure, 2012.

[13] DAVIS, A. Lessons learned from 50 bugs: Common USB driver vulnerabilities. `https://www.nccgroup.trust/globalassets/our-research/uk/whitepapers/usb_driver_vulnerabilities_whitepaper_v2.pdf`, 2013.

[14] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. A. Automated whitebox fuzz testing. In *Proc. Network and Distributed System Security Symp. (NDSS)* (2008), The Internet Society.

[15] GOODSPEED, T., AND BRATUS, S. Emulating USB devices with python. `https://travisgoodspeed.blogspot.de/2012/07/emulating-usb-devices-with-python.html`, September 2012.

[16] JANG, Y., LEE, S., AND KIM, T. Breaking kernel address space layout randomization with Intel TSX. In *ACM Conf. Computer and Communications Security (CCS)* (2016), ACM, pp. 380–392.

[17] JARBOUI, T., ARLAT, J., CROUZET, Y., AND KANOUN, K. Experimental analysis of the errors induced into Linux by three fault injection techniques. In *Proc. Int. Conf. Dependable Systems and Networks (DSN)* (2002), pp. 331–336.

[18] JODEIT, M., AND JOHNS, M. USB device drivers: A stepping stone into your kernel. In *Proc. European Conf. Computer Network Defense (EC2ND)* (2010), IEEE, pp. 46–52.

[19] KUZNETSOV, V., CHIPOUNOV, V., AND CANDEA, G. Testing closed-source binary device drivers with DDT. In *USENIX Annual Technical Conference* (2010), USENIX Association.

[20] LTP CONTRIBUTORS. Linux Test Project. `https://linux-test-project.github.io/`, 2017.

[21] MARINESCU, P. D., AND CANDEA, G. LFI: A practical and general library-level fault injector. In *Proc. Int. Conf. Dependable Systems and Networks (DSN)* (2009), pp. 379–388.

[22] MASKIEWICZ, J., ELLIS, B., MOURADIAN, J., AND SHACHAM, H. Mouse trap: Exploiting firmware updates in USB peripherals. In *USENIX Workshop on Offensive Technologies (WOOT)* (2014).

[23] NATELLA, R., COTRONEO, D., AND MADEIRA, H. Assessing dependability with software fault injection: A survey. *ACM Comput. Surv. 48*, 3 (2016), 44:1–44:55.

[24] PAX GRSECURITY. RAP: RIP ROP. `http://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf`, 2017.

[25] PRASAD, V., COHEN, W., EIGLER, F. C., HUNT, M., KENISTON, J., AND CHEN, B. Locating system problems using dynamic instrumentation. In *Proc. Ottawa Linux Symposium (OLS)* (2005), pp. 49–64.

[26] SCHUMILO, S., SPENNEBERG, R., AND SCHEWARTKE, H. Don't trust your USB! How to find bugs in USB device drivers. In *Blackhat Europe* (2014).

[27] USB IMPLEMENTORS FORUM. USB 3.1 Specification. `http://www.usb.org/developers/docs`, 2017.

[28] VAN TONDER, R., AND ENGELBRECHT, H. A. Lowering the USB fuzzing barrier by transparent two-way emulation. In *USENIX Workshop on Offensive Technologies (WOOT)* (2014).