Homework Assignment 2

Robert Bland and Tyler Townsend

November 7, 2018

1 Tower Probe Race condition: CVE-2017-15102

1.1 Problem

The tower_probe function has a security vulnerability. This is vulnerability is exposed when a write operation happens concurrently with a tower_delete, which is called and derefences a pointer that will be used later with the write operation. This scenario is a race condition and has to be set up in a distinct series of operations discussed in the next subsection. One requirement is to delay the control message sent in the probe function in legousbtower.c. This is done between the registering of the interface and the reading of the boards firmware ID. Another requirement for this vulnerability is that the 0 address must be mappable on the machine. This allows it to be possible to create a local privilege escalation exploit using a write-what-where condition. This exploit is caused by a remapping of the dev-interrupt_out_buffer in tower_write operation that was called via a NULL dereference. This means that is now possible to run any malicious scripts from that USB on the computer.

1.2 Existence

This vulnerability has existed since 2003 and just recently got patched in March 2018, although this did have solutions readily available last year with a simple change in structure[3]. The versions it ranges from version 2.6.12.1 up to 4.8.11 and exists on Redhat Enterprise Linux version 5 & version 6[1]. The severity of this problem was listed as a 6.9 since the requirements to do it needed the person to physically plug it in, but if they managed to do this then they could run any script using that USB[1].

1.3 Exact Location and Execution of Problem

One thing to note before we start assumptions is that not all of the code for tower_probe is shown below, but it's enough to show the problem at hand. Now we will assume that we have tampered the USB's firmware ID to throw an error later and that we will run a write operation concurrently later as well. We execute tower_probe until we finish registering our USB device on line 22. Then we delay the thread that is running the probe function before it reaches line 25. Then the write operation mentioned earlier will initiate a write to the device file. We stop stalling tower_probe and line 25 will throw an error because of the ID and call

Tower_delete. This does not stop the execution of this write operation and tower_delete will free dev->interrupt_out_urb on line 7. Now the write operation will throw a NULL derefence at line 6 and 18 and will remap dev->interrupt_out_urb to 0. This now is a write-what-where condition and creates a local privilege escalation exploit which then allows us to run any malicious code.

```
@@ -886,24 +886,6 @@ static int tower_probe (struct usb_interface *interface,
    const struct usb_device
...
  dev->interrupt_in_interval = interrupt_in_interval ? interrupt_in_interval :
    dev->interrupt_in_endpoint->bInterval;
  dev->interrupt_out_interval = interrupt_out_interval ?
   interrupt_out_interval : dev->interrupt_out_endpoint->bInterval;

  /* we can register the device now, as it is ready */
  usb_set_intfdata (interface, dev);

  retval = usb_register_dev (interface, &tower_class);

  if (retval) {
    /* something prevented us from registering this driver */
    dev_err(idev, "Not able to get a minor for this device.\n");
    usb_set_intfdata (interface, NULL);
    goto error;
  }
  dev->minor = interface->minor;

  /* let the user know what node this device is now attached to */
  dev_info(&interface->dev, "LEGO USB Tower #%d now attached to major "
     "%d minor %d\n", (dev->minor - LEGO_USB_TOWER_MINOR_BASE),
     USB_MAJOR, dev->minor);

  /* get the firmware version and log it */
  result = usb_control_msg (udev,
          usb_rcvctrlpipe(udev, 0),
@@ -924,6 +906,23 @@ static int tower_probe (struct usb_interface *interface,
    const struct usb_device
     get_version_reply.minor,
     le16_to_cpu(get_version_reply.build_no));

 exit:
  return retval;
```

Figure 1.1: `tower_probe()`

```
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);
}
```

Figure 1.2: `tower_delete()`

```
static ssize_t tower_write  (struct file *file, const char __user *buffer,
    size_t count, loff_t *ppos)
{
  ...

  /* send off the urb */
  usb_fill_int_urb(dev->interrupt_out_urb,
       dev->udev,
       usb_sndintpipe(dev->udev, dev->interrupt_out_endpoint->bEndpointAddress
    ),
       dev->interrupt_out_buffer,
       bytes_to_write,
       tower_interrupt_out_callback,
       dev,
       dev->interrupt_out_interval);

  dev->interrupt_out_busy = 1;
  wmb();

  retval = usb_submit_urb (dev->interrupt_out_urb, GFP_KERNEL);
  if (retval) {
    dev->interrupt_out_busy = 0;
    dev_err(&dev->udev->dev,
      "Couldn't submit interrupt_out_urb %d\n", retval);
    goto unlock_exit;
  }

  ...
  unlock_exit:
  /* unlock the device */
  mutex_unlock(&dev->lock);

exit:
  return retval;
}
}
```

Figure 1.3: `tower_write()`

1.4 Solution

The solution is surprisingly simple since it only requires us to change where we register the USB interface[3]. Normally the program registers the USB interface then checks the firmware ID, but in the solution we move the code for registering the USB after we check the firmware ID. This makes it impossible for a user to use the USB and do a write operation before confirming the ID, thus eliminating the race-condition. As shown in the code below, the

lines with the negative symbols are the lines we delete and the lines with the plus symbols are the ones where we add to the code and it is a simple cut and paste. The + indicate added lines and the − indicate removed lines.

```
@@ -886,24 +886,6 @@ static int tower_probe (struct usb_interface *interface,
    const struct usb_device
...
  dev->interrupt_in_interval = interrupt_in_interval ? interrupt_in_interval :
    dev->interrupt_in_endpoint->bInterval;
  dev->interrupt_out_interval = interrupt_out_interval ?
   interrupt_out_interval : dev->interrupt_out_endpoint->bInterval;

- /* we can register the device now, as it is ready */
- usb_set_intfdata (interface, dev);
-
- retval = usb_register_dev (interface, &tower_class);
-
- if (retval) {
-   /* something prevented us from registering this driver */
-   dev_err(idev, "Not able to get a minor for this device.\n");
-   usb_set_intfdata (interface, NULL);
-   goto error;
- }
- dev->minor = interface->minor;
-
- /* let the user know what node this device is now attached to */
- dev_info(&interface->dev, "LEGO USB Tower #%d now attached to major "
-    "%d minor %d\n", (dev->minor - LEGO_USB_TOWER_MINOR_BASE),
-    USB_MAJOR, dev->minor);
-
  /* get the firmware version and log it */
  result = usb_control_msg (udev,
          usb_rcvctrlpipe(udev, 0),
@@ -924,6 +906,23 @@ static int tower_probe (struct usb_interface *interface,
    const struct usb_device
      get_version_reply.minor,
      le16_to_cpu(get_version_reply.build_no));

+ /* we can register the device now, as it is ready */
+ usb_set_intfdata (interface, dev);
+
+ retval = usb_register_dev (interface, &tower_class);
+
+ if (retval) {
+   /* something prevented us from registering this driver */
+   dev_err(idev, "Not able to get a minor for this device.\n");
+   usb_set_intfdata (interface, NULL);
+   goto error;
+ }
+ dev->minor = interface->minor;
+
+ /* let the user know what node this device is now attached to */
+ dev_info(&interface->dev, "LEGO USB Tower #%d now attached to major "
+    "%d minor %d\n", (dev->minor - LEGO_USB_TOWER_MINOR_BASE),
```

```
47 +     USB_MAJOR, dev->minor);
48
49  exit:
50   return retval;
```

Figure 1.4: Solution to `tower_probe()`

1.5 Concurrency

This is related to concurrency since this problem only exists since we can do a read/write operation while the program is registering the USB and checking the board firmware ID. If this was executed in a sequential manner then delaying the probe function would be pointless since it will not be able to execute anything else until this delay is done. Which will eventually throw an error and cause the queued write operation to never happen since the device will be unregistered before it can write. This portrays just how serious race conditions are in a concurrent environment and how small changes or certain conditions can allow attackers to run malicious code. These conditions are very spread out and hard to find, but can exist for over a decade before it even gets patched out. This is pretty terrifying knowing that concurrent programs can have unexpected outcomes that lead to more than just mismatched data and incorrect sequential outcomes, since this outcome let the attacker do anything they want. Preventive measures for this scenario would have been to reconsider the ordering of registering a device and checking it's firmware. Clearly if we have an operation that can nullify the work done in the previous lines of code, then it would be more optimal to check that scenario first then to continue with the rest of the program.

2 Driver_override: CVE-2017-12146

2.1 Problem Description

The vulnerability of the driver_override implementation in drivers/base/platform.c in the Linux kernel (before 4.12.1) is susceptible to race condition when different threads are reading vs storing a different driver override [**?**]. The race condition allows local users to gain elevated privileges by leveraging the race condition. A local attacker could use this t o possibly gain administrative privileges[**?**].

2.2 History

The weakness was released 09/08/2017 and traded as CVE-2017-12146 since 08/01/2017. The attack needs to be approached locally, and successful exploitation needs a single authentication. With known technical details, no exploit is available[**?**]. A possible race condition occurs between driver_override_store() and driver_override_show().

The driver_override patch was a result to allow the driver for a device to be specified which would override standard OF, ACPI, ID table and name matching. When specified, if a driver has a name matching the driver_override, the driver will have an opportunity to bind to the device which is specified by a writing a string to the driver_override file. This is

done in the `driver_override_store()` function as seen in figure 2.1. To note, any device that is represent as platform device(i.e `struct platform_device`) allows them to connect to a virtual "platform bus" that allows the kernel to communicate with devices via a specified driver that are not necessarily “plug and play” [**?**].

The current driver will not automatically unbind from the device from writing to driver_override nor make any attempt to automatically load the specified driver. If no driver with a matching name is currently loaded in the kernel, the device will not bind to any driver. This also allows devices to opt-out of driver binding using a driver_override name such as "none".

Platform devices have long been used in this role in the kernel. So now we have a driver for a platform device, but no actual devices yet. As we noted at the beginning, platform devices are inherently not discoverable so there must be a another way to tell the kernel about their existence. That is typically done with the creation of a static `platform_device` structure providing, at a minimum, a name which is used to find the associated driver. The platform.c file provides a platform 'pseudo' bus for legacy devices[**?**] (i.e. for devices that are not “plug and play”).

2.3 Exact Location and Execution of Problem

The problem occurs due to shared memory between two threads, one which can be calling a store() and the other a show(). In Figure 2.1 The race condition occurs from calling `pdev->driver_override` on lines 19 to 24. Show() requires access to the same data at lines 4 - 6 in Figure 2.2. If `sprintf()` does not finish writing to `buf` before the memory storing `pdev->driver_override` is freed, this will cause corruption.

Suppose there are two threads T_1, and T_2 are performing a store(), show() operation, respectively. Thread T_1 reads `old=pdev->driver_override`. Then T_2 calls `sprintf(buf, "%s\n", pdev->driver_override)`. When `sprintf()` starts reading from `pdev->driver_override`, Thread T_1 executes `pdev->driver_override = driver_override` followed by kfree(old) corrupting thread T_2's execution.

Similarly, there could be a data race between multiple threads performing a store() where memory leaks are possible[**?**].

```
1  static ssize_t driver_override_store(struct device *dev,
2              struct device_attribute *attr,
3              const char *buf, size_t count)
4  {
5    struct platform_device *pdev = to_platform_device(dev);
6    char *driver_override, *old = pdev->driver_override, *cp;
7
8    if (count > PATH_MAX)
9      return -EINVAL;
10
11   driver_override = kstrndup(buf, count, GFP_KERNEL);
12   if (!driver_override)
13     return -ENOMEM;
14
15   cp = strchr(driver_override, '\n');
16   if (cp)
17     *cp = '\0';
```

```
18
19    if (strlen(driver_override)) {
20      pdev->driver_override = driver_override;
21    } else {
22      kfree(driver_override);
23      pdev->driver_override = NULL;
24    }
25
26    kfree(old);
27
28    return count;
29  }
```

Figure 2.1: `driver_override_store()`

```
1  static ssize_t driver_override_show(struct device *dev,
2              struct device_attribute *attr, char *buf)
3  {
4    struct platform_device *pdev = to_platform_device(dev);
5
6    return sprintf(buf, "%s\n", pdev->driver_override);
7  }
```

Figure 2.2: `driver_override_show()`

2.4 Solution

The solution requires adding a lock to avoid race condition. The lock tries to protect against a race between store and show. In Figure 2.3, the first change is made to set `old=pdev->driver_override` before modifying `pdev->driver_override`. To provide atomicity, the device , `dev`, from which `pdev` was called is locked. Therefore, this will provide a means of mutual exclusion. The threads then finish and unlock as seen in lines 5 and 13.

```
1  static ssize_t driver_override_store(struct device *dev,
2    if (cp)
3      *cp = '\0';
4
5  + device_lock(dev);
6  + old = pdev->driver_override;
7    if (strlen(driver_override)) {
8      pdev->driver_override = driver_override;
9    } else {
10     kfree(driver_override);
11     pdev->driver_override = NULL;
12   }
13 + device_unlock(dev);
14
15   kfree(old);
```

Figure 2.3: Solution to `driver_override_store()`

To safeguard show(), the same method is taken and a lock is used as seen in Figure

2.4. The lock prevents an interrupt during the execution of `sprintf()`. Therefore, only one thread may be able to read and write to the device's driver_override.

```
static ssize_t driver_override_show(struct device *dev,
            struct device_attribute *attr, char *buf)
{
  struct platform_device *pdev = to_platform_device(dev);
+ ssize_t len;

- return sprintf(buf, "%s\n", pdev->driver_override);
+ device_lock(dev);
+ len = sprintf(buf, "%s\n", pdev->driver_override);
+ device_unlock(dev);
+ return len;
}
```

Figure 2.4: Solution to `driver_override_show()`

2.5 Concurrency and Synchronization

Due to the lock used on the device, this creates a bottleneck of contention for the lock if many threads are competing for it. These two function now have a sequential behavior and are no longer concurrent, which is most likely a good thing. However, the behavior is now linearizable and overlapping events are sequentially consistent and well defined. Because of the lock, there is no possibility of the memory pointed to by `pdev->driver_override` changing while show() is writing to a buffer. Using the lock ensures that if the thread reading the `driver_override` starts then it will atomically and will write whatever it expects.

References

[1] https://nvd.nist.gov/vuln/detail/CVE-2017-15102

[2] Tower_write and tower_delete take from https://github.com/torvalds/linux/blob/master/drivers/usb/misc/legousbtower.c

[3] Solution taken from https://github.com/torvalds/linux/commit/2fae9e5a7babada041e2e161699ade2447a01989

[4] https://people.canonical.com/ ubuntu-security/cve/2017/CVE-2017-12146.html

[5] https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2017-12146

[6] https://elixir.bootlin.com/linux/v4.12.14/source/drivers/base/platform.c

[7] https://vuldb.com/?id.106296

[8] https://www.redhat.com/archives/libvir-list/2014-April/msg00382.html

[9] https://lore.kernel.org/patchwork/patch/783378/