Tower Probe Race Condition

Presented By Robert Bland & Tyler Townsend

What is a Tower_Probe?

- Tower_Probe is a function that exists within the legousbtower driver
  - Legousbtower driver provides support for LegoZ Mindstorms USB IR Tower
- The Tower_Probe function is responsible for both registering the usb device and for confirming the devices firmware board ID
- If there is an error in confirming firmware ID then Tower_Probe will call Tower_Delete (Important)

What is Legousbtower?

Device driver built for Lego USB IR Device

- First released in 2001
- Added to the linux kernel in version 2.6.1

Problem at Hand (Assumptions)

We must assume a few the attacker has done the following:

1. The attacker has a forged USB device with an invalid firmware ID
2. Will do a write operation using this device
3. Will delay the call to Tower_Delete until the write operation starts

Problem at Hand (Effects)

What can the attacker do now that we have this scenario?

- Now possible to create a race condition between the write operation and Tower_Probe executing Tower_Delete
- The race condition is possible because the device is registered before confirming the firmware ID
  - Allows attacker to perform global reads/writes before calling the ID confirm operation
- Bad firmware ID causes the ID confirm operation inside the Tower_Probe to call Tower_Delete

Problem (Picture)

Lines 5-21 is when Tower_Probe registers the device

We stall before calling usb_control_msg on line 24

Attacker concurrently executes a read/write operation and then stop stalling to allow for the Tower_Delete call

```
1  @@ -886,24 +886,6 @@ static int tower_probe (struct usb_interface *
     interface, const struct usb_device
2    dev->interrupt_in_interval = interrupt_in_interval ?
       interrupt_in_interval : dev->interrupt_in_endpoint->bInterval;
3    dev->interrupt_out_interval = interrupt_out_interval ?
       interrupt_out_interval : dev->interrupt_out_endpoint->bInterval
       ;
4
5    /* we can register the device now, as it is ready */
6    usb_set_intfdata (interface, dev);
7
8    retval = usb_register_dev (interface, &tower_class);
9
10   if (retval) {
11     /* something prevented us from registering this driver */
12     dev_err(idev, "Not able to get a minor for this device.\n");
13     usb_set_intfdata (interface, NULL);
14     goto error;
15   }
16   dev->minor = interface->minor;
17
18   /* let the user know what node this device is now attached to */
19   dev_info(&interface->dev, "LEGO USB Tower #%d now attached to
       major "
20      "%d minor %d\n", (dev->minor - LEGO_USB_TOWER_MINOR_BASE),
21      USB_MAJOR, dev->minor);
22
23   /* get the firmware version and log it */
24   result = usb_control_msg (udev,
25           usb_rcvctrlpipe(udev, 0),
26 @@ -924,6 +906,23 @@ static int tower_probe (struct usb_interface *
     interface, const struct usb_device
27      get_version_reply.minor,
28      le16_to_cpu(get_version_reply.build_no));
29
30  exit:
31   return retval;
```

Result

```
291  static inline void tower_delete (struct lego_usb_tower *dev)
292  {
293          tower_abort_transfers (dev);
294
295          /* free data structures */
296          usb_free_urb(dev->interrupt_in_urb);
297          usb_free_urb(dev->interrupt_out_urb);
298          kfree (dev->read_buffer);
299          kfree (dev->interrupt_in_buffer);
300          kfree (dev->interrupt_out_buffer);
301          kfree (dev);
302  }
```

```
681          if (copy_from_user (dev->interrupt_out_buffer, buffer, bytes_to_write)) {
682                  retval = -EFAULT;
683                  goto unlock_exit;
684          }
685
686          /* send off the urb */
687          usb_fill_int_urb(dev->interrupt_out_urb,
688                           dev->udev,
689                           usb_sndintpipe(dev->udev, dev->interrupt_out_endpoint->bEndpointAddress),
690                           dev->interrupt_out_buffer,
691                           bytes_to_write,
692                           tower_interrupt_out_callback,
693                           dev,
694                           dev->interrupt_out_interval);
695
696          dev->interrupt_out_busy = 1;
697          wmb();
698
699          retval = usb_submit_urb (dev->interrupt_out_urb, GFP_KERNEL);
700          if (retval) {
701                  dev->interrupt_out_busy = 0;
702                  dev_err(&dev->udev->dev,
703                          "Couldn't submit interrupt_out_urb %d\n", retval);
704                  goto unlock_exit;
705          }
706          retval = bytes_to_write;
707
708  unlock_exit:
709          /* unlock the device */
710          mutex_unlock(&dev->lock);
711
712  exit:
713          return retval;
```

Delete frees dev->interrupt_out_urb (Line 297)

Write operation then has a NULL pointer dereference and causes a write-what-where condition

Result (Part 2)

The following is what occurs:

1. Exposes a write-what-where condition by remapping dev->interrupt_out_buffer
    a. Write-what-where condition is when the attacker can write an arbitrary value to an arbitrary location, usually caused by overflow
2. Leads to local privilege escalation and allows the attacker to execute their own malicious code

Note: This is only possible if 0 is mappable on the Linux machine and the linux machine kernel has to be a version between 2.6.1x and 4.8.0x

Solution

Solution is fairly simple

It’s only a restructuring of the already existing code in Tower_Probe

Instead of registering the device before confirming board’s ID we register it after the confirmation

- Makes stalling meaningless by eliminating the possibility of a read/write operation to happen concurrently with a Tower_Delete

Solution (Picture)

What to know about the picture:

- Negative signs are the lines of code that we delete
- Positive signs are the lines of code we add

Note that we cut and pasted where we register our device to be after we check the firmware ID

```
@@ -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),
+    USB_MAJOR, dev->minor);

 exit:
  return retval;
```

Concurrency and you

What does this have to do with concurrency? There was no locks or compare-and-swaps shown in the code, so how does it relate to it?

- This vulnerability is only present with concurrent operations
  - A sequential ordering would cause the stalling to be pointless and eliminates race conditions
    - Impossible to do a read/write operation and call Tower_Delete
- Exemplifies how race conditions in concurrent operations can cause security vulnerabilities given the right conditions
  - Conditions are very specific, but a vulnerability is still a security risk that has to be addressed
  - If it can happen once then it can be exploited repeatedly

References

Github with change - https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=2fae9e5a7babada041e2e161699ade2447a01989

Link to vulnerability history - https://nvd.nist.gov/vuln/detail/CVE-2017-15102#vulnCurrentDescriptionTitle

LegoUSB Project website - http://legousb.sourceforge.net/legousbtower/index.shtml