# Exploiting USB/IP in Linux
## UBOAT or CVE-2016-3955

Ignat Korchagin (ignat@cloudflare.com)

## Introduction

UBOAT (USB/IP Buffer Overflow Attack) is a vulnerability in USB over IP framework (USB/IP), which is part of Linux kernel code. This framework was originally developed by USB/IP project[1] and merged into mainline Linux kernel source tree from version 3.17 and allows hardware to share connected USB devices over IP network: devices, connected to USB/IP server, appear on the client as if they were plugged in locally.

UBOAT allows an attacker to write arbitrary data to Linux kernel memory heap on USB/IP client, possibly causing denial of service (DoS) or arbitrary code execution at privileged levels.

Paper outline:

- What is USB/IP
- USB/IP implementation in Linux
- Vulnerable USB/IP code
- Potential impact
- Hardening USB/IP setups

## What is USB/IP

From project website[1]:

"*The USB/IP Project aims to develop a general USB device sharing system over IP network. To share USB devices between computers with their full functionality, USB/IP encapsulates "USB I/O messages" into TCP/IP payloads and transmits them between computers. Original USB device drivers and applications can be also used for remote USB devices without any*

*modification of them. A computer can use remote USB devices as if they were directly attached;...*".

General design of the solution is outlined on **Figure 1**.

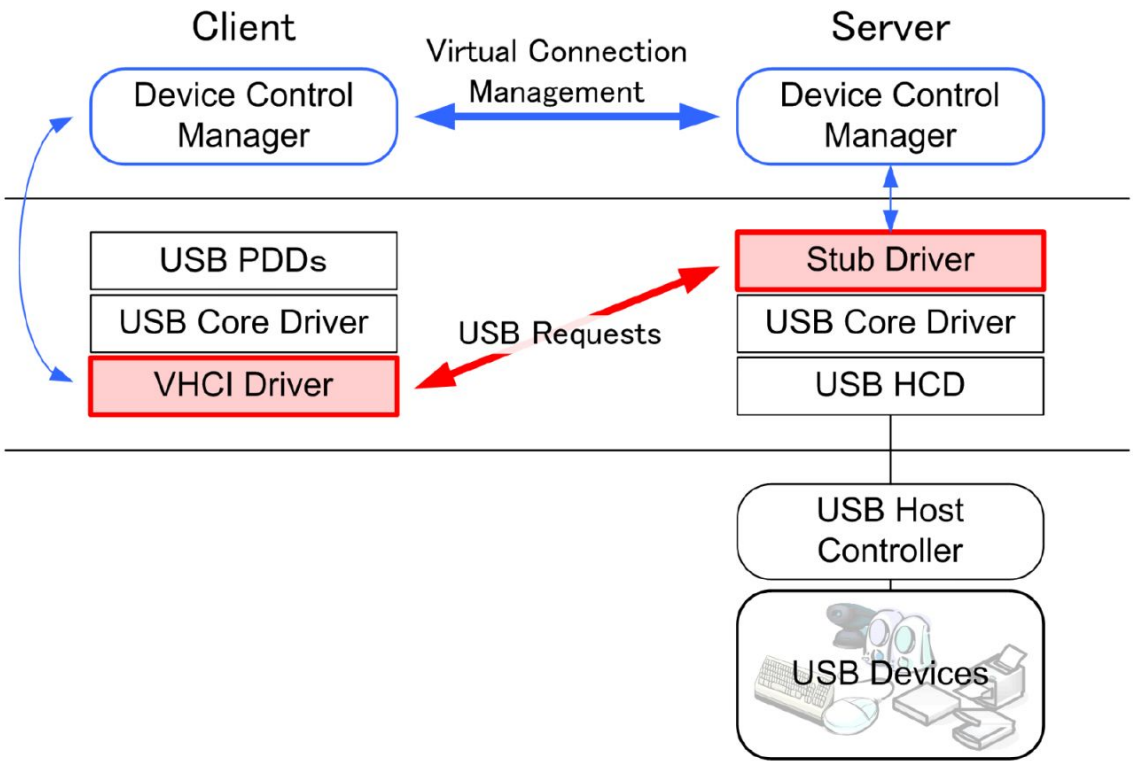


Fig. 1 USB/IP design

The project website[1] and related paper[2] have detailed explanation of the architecture and implementation of the USB/IP framework. We will just describe major high level components and roles:

Roles:

- *server* - has physical USB devices connected and "exports" them over network connection
- *client* - connects to the server and "imports" devices over the network (devices appear to be as plugged in directly to the client)

Basically, the framework consists of these components:

- *USB stub driver* - runs on the server, intercepts all USB request blocks (*URBs*) from the physical device and transmits them over the network, if configured
- *USB virtual host controller* - runs on the client, receives URBs from the network and passes them to local USB subsystem and appropriate device drivers
- *userspace helper tools* - run on client and server, establish a network connection between client and server, handle configuration etc

## USB/IP implementation in Linux

USB/IP code and components are part of the Linux mainline source tree since version 3.17. USB/IP kernel code resides in *drivers/usb/usbip* subdirectory and userspace helper tools are in *tools/usb/usbip.* Most popular Linux distributions have USB/IP kernel code compiled as modules and provide them as part of distribution kernel standard package.

USB/IP basic kernel modules:

Described below kernel modules may have different names depending on the major version of the respective kernel source tree. Also, newer versions of Linux may contain additional modules. However, the core kernel components, which are relevant to basic USB/IP functionality are as follows (as of Linux 4.4 source tree):

- *usbip-host.ko* - module to support server-side USB/IP functionality, implements USB stub driver described above
- *vhci-hcd.ko* - module to support client-side USB/IP functionality, implements USB virtual host controller
- *usbip-core.ko* - module, which implements some common code for the above modules

USB/IP basic userspace tools:

These tools are used to setup and configure USB/IP on client and server and implement primary CLI interface to USB/IP framework in Linux. These tools include:

- *usbipd* - a daemon, which runs on the server and accepts connections from USB/IP clients
- *usbip* - a CLI tool, which is used by users to enable/configure USB/IP, which may:
    - communicate with local instance of usbipd and configure devices to be exported on a particular USB/IP server
    - communicate with remote USB/IP servers and actually import USB devices from them

## Sharing a USB device

Suppose USB/IP server was configured to export one or more USB devices, the client may invoke two basic operations: list exported devices from a server and actually import a device from the list. Listing exported devices does not involve any kernel code: under the hood the client just establishes a TCP connection to the server, sends a request for the list according to USB/IP network protocol[3]. The server (usbipd) serves the list in the response from its memory without going to the kernel and both sides close the TCP stream.

When client requests to import a USB device, the following set of operations happen:

1. Client opens a TCP connection to the server and sends a request to import a USB device according to USB/IP network protocol[3].
2. The server receives the response, verifies that requested device is still configured to be exported and acknowledges the request.
3. Both sides keep the TCP connection open and pass file descriptors associated with the socket representing the connection to the Linux kernel.
4. Respective modules in the kernel (*usbip-host.ko* and *vhci-hcd.ko*) start forwarding URBs between USB subsystems over the TCP connection, encapsulating them in USB/IP “messages” according to USB/IP network protocol[3].

So unlike conceptually similar subsystems (like *TUN/TAP virtual network driver* for example), where kernel just exposes some low-level structure or blob to userspace and userspace is responsible to deliver it appropriately, USB/IP takes a bit different approach by implementing the application layer networking protocol in the kernel directly.

## Vulnerable USB/IP code

USB/IP network protocol[3] is rather simple: a single USB/IP “message” consists of a header with some metadata (including length) and the actual URB to be forwarded to the other party’s USB subsystem. Since, as described above, the actual protocol is implemented directly in the Linux kernel, potential security implications are much higher for not properly validating external input. Unfortunately, this is the case with USB/IP implementation. The vulnerability is located in the receiving code path for the client. Below is a part of USB/IP high level receive function on the client side (*drivers/usb/usbip/vhci_rx.c*):

```
static void vhci_recv_ret_submit(struct vhci_device *vdev,
                                 struct usbip_header *pdu)
{
...
    /* unpack the pdu to a urb */
    usbip_pack_pdu(pdu, urb, USBIP_RET_SUBMIT, 0);

    /* recv transfer buffer */
    if (usbip_recv_xbuff(ud, urb) < 0)
        return;

    /* recv iso_packet_descriptor */
    if (usbip_recv_iso(ud, urb) < 0)
        return;
…
```

The code above includes two steps: parsing received USB/IP header (*usbip_pack_pdu*) and receiving the rest of the URB data from the network (*usbip_recv_xbuff*). These functions are implemented in the common for client and server USB/IP kernel module (*usbip-core.ko*, *drivers/usb/usbip/usbip_common.c*):

```
static void usbip_pack_ret_submit(struct usbip_header *pdu, struct urb *urb,
                  int pack)
{
    struct usbip_header_ret_submit *rpdu = &pdu->u.ret_submit;

    if (pack) {
        rpdu->status        = urb->status;
        rpdu->actual_length    = urb->actual_length;
        rpdu->start_frame    = urb->start_frame;
        rpdu->number_of_packets = urb->number_of_packets;
        rpdu->error_count    = urb->error_count;
    } else {
        urb->status        = rpdu->status;
        urb->actual_length    = rpdu->actual_length;
        urb->start_frame    = rpdu->start_frame;
        urb->number_of_packets = rpdu->number_of_packets;
        urb->error_count    = rpdu->error_count;
    }
```

```
}

void usbip_pack_pdu(struct usbip_header *pdu, struct urb *urb, int cmd,
            int pack)
{
    switch (cmd) {
    case USBIP_CMD_SUBMIT:
        usbip_pack_cmd_submit(pdu, urb, pack);
        break;
    case USBIP_RET_SUBMIT:
        usbip_pack_ret_submit(pdu, urb, pack);
        break;
    default:
        /* NOT REACHED */
        pr_err("unknown command\n");
        break;
    }
}
...
int usbip_recv_xbuff(struct usbip_device *ud, struct urb *urb)
{
    int ret;
    int size;

    if (ud->side == USBIP_STUB) {
        /* the direction of urb must be OUT. */
        if (usb_pipein(urb->pipe))
            return 0;

        size = urb->transfer_buffer_length;
    } else {
        /* the direction of urb must be IN. */
        if (usb_pipeout(urb->pipe))
            return 0;

        size = urb->actual_length;
    }

    /* no need to recv xbuff */
    if (!(size > 0))
        return 0;

    ret = usbip_recv(ud->tcp_socket, urb->transfer_buffer, size);
    if (ret != size) {
        dev_err(&urb->dev->dev, "recv xbuf, %d\n", ret);
        if (ud->side == USBIP_STUB) {
            usbip_event_add(ud, SDEV_EVENT_ERROR_TCP);
        } else {
            usbip_event_add(ud, VDEV_EVENT_ERROR_TCP);
            return -EPIPE;
        }
    }

    return ret;
}
```

We see that *usbip_pack_pdu* calls *usbip_pack_ret_submit* on response path, which on unpack operation just puts the received *actual_length* from the server into the kernel urb structure. Later, the code in *usbip_recv_xbuff* uses this *urb->actual_length* to receive the actual URB data and place it in the *urb->transfer_buffer*. The problem here is that there

are no checks, that the received previously *urb->actual_length* can fit in the *urb->transfer_buffer.* So, by crafting a special USB/IP response to the client (either by modifying USB/IP packets on the network or compromising the USB/IP server) we can write arbitrary size data to the Linux kernel memory beyond *urb->transfer_buffer.*

## Potential impact

In Linux USB subsystem URBs are usually allocated either by core USB code or USB device drivers for a specific device. While basic URB structure can reside either on the stack or allocated dynamically, from the kernel heap, *urb->transfer_buffer* is allocated mostly from the heap. The size of the allocation (and therefore respective SLUB cache, from which the allocated chunk comes) depends on the "intention" of the URB: in USB world each device driver is "expected" to know the potential amount it can receive from its device, so the driver allocates the buffers accordingly to accommodate this maximum.

It is worth to note that low level USB communication is always performed from host to device, so usually it is not possible to get the size of the actual response first, allocate the buffer accordingly and "read" the response - buffers are always preallocated with some "maximum" size in mind.

Since with the above vulnerability it is possible to write almost any data past *urb->transfer_buffer* boundary an attacker may try to:

- crash the target system (DoS)
- inject and/or modify kernel dynamic data
- change kernel execution path (code execution)

While the first attack (DoS) is relatively easy, data injection and/or code execution from kernel heap might be more complicated, but still possible. This paper will not describe all the details, but rather refer to already published techniques, which were developed for other similar discovered vulnerabilities, such as "Linux Kernel CAN SLUB Overflow"[4].

Abovementioned post describes a way to exploit the overflow in 96-byte SLUB cache. It is worth pointing out, that with UBOAT it is possible for an attacker to control which SLUB cache size to exploit: since the attacker can modify incoming USB/IP traffic at will, he/she can emulate any USB device supported by the Linux kernel. Therefore, the attacker only

needs to find a device driver in the Linux source tree, which allocates transfer buffers for its URBs from the desired SLUB cache.

## Hardening USB/IP setups

Even without the vulnerability USB/IP is very powerful, but dangerous technology: it allows emulating almost any USB device remotely from the network. Unfortunately core USB/IP tools neither support almost any security features (apart from potential capability of compiling with TCP Wrapper library, which provides limited access control features) nor provide any security considerations in setting up the framework. Below are simple recommendations on how to secure a USB/IP setup:

- often patch your system - since USB/IP is mostly implemented in Linux kernel it is vital to keep the code up to date and therefore free of discovered bugs, because they may lead to vulnerabilities
- protect your traffic - while it is applicable to almost any system, which uses network, USB/IP is very sensitive to network attacks, since it exposes very low-level functionality over the network, so it is important to secure all USB/IP network connections (for example, with IPSec or TLS) and ensure integrity and confidentiality of transmitted and received data
- ensure your USB/IP server is trustworthy - since USB/IP server partly controls client's USB subsystem remotely, one needs to ensure authenticity of the server and take relevant measures to make sure the access to the server is sufficiently restricted and server is running up-to-date operating system and software

## References


1. "USB/IP Project." http://usbip.sourceforge.net/
2. T. Hirofuchi, E. Kawai, K. Fujikawa, H. Sunahara: USB/IP: A Transparent Device Sharing Technology over IP Network, *IPSJ Digital Courier 1:394-406 · January 2005*
3. "USB/IP protocol" https://www.kernel.org/doc/Documentation/usb/usbip_protocol.txt
4. "Linux Kernel CAN SLUB Overflow | Jon Oberheide." 10 Sep. 2010, https://jon.oberheide.org/blog/2010/09/10/linux-kernel-can-slub-overflow/