Making USB Great Again with USBFILTER

Dave Tian*, Nolen Scaife*, Adam Bates**, Kevin Butler*, Patrick Traynor*

*University of Florida, Gainesville, FL
**University of Illinois, Urbana-Champaign, IL

USENIX Security’16, Austin, TX
Aug 11, 2016
Make it real...
Why USB was great

• Universal Serial Bus
  • USB 1.0/2.0/3.0/3.1/Type-C
• Speed
  • 10 gigabits per second
• Ubiquitous
Why USB is not great anymore
USB enumeration

Host

SetAddress(n)

ACK

GetDescriptor(Device)

MNF: Kingston, Product: Flash Drive

GetDescriptor(Interface)

Storage

Device

Human Interface
USB Device

The diagram illustrates a USB device with multiple interfaces and endpoints. Each interface is identified with a number (Interface 0, Interface 1, Interface 2), and endpoints are labeled as EP0, EP1, EP2, and EPn. The endpoints are categorized as In and Out, indicating the direction of data transmission. The device structure is designed to facilitate communication between the USB controller and the device, allowing for various functionalities such as data transfer, control, and synchronization.
USB packet

USB_packet("key")

USB_packet("data")

USB_packet("key")
USBFILTER implements a USB-layer reference monitor within the kernel, by filtering USB packets to different USB devices to control the communications between applications and devices based on rules configured.

We consider the following actions by an adversary:

- **Device Tampering**: The adversary may attempt to attach or tamper with a previously-authorized device to add unauthorized functionality (e.g., BadUSB [27]).
- **Unauthorized Devices**: Unauthorized devices attached to the system either physically or virtually [21] can be used to discreetly interact with the host system or to provide data storage for future exfiltration.
- **Unauthorized Access**: The adversary may attempt to enable or access authorized devices on a host (e.g., webcam, microphone, etc.) via unauthorized software to gain access to information or functionality that would otherwise inaccessible.

We assume that as a kernel component, the integrity of USBFILTER depends on the integrity of the operating system and the host hardware (except USB devices). Code running in the kernel space has unrestricted access to the kernel's memory, including our code, and we assume that the code running in the kernel will not tamper with USBFILTER. We discuss how we ensure runtime and platform integrity in our experimental setup in Section 3.4.

### 3.2 Design Goals

Inspired by the Netfilter [40] framework in the Linux kernel, we designed USBFILTER to enable administrator-defined rule-based filtering for the USB protocol. To achieve this, we first designed our system to satisfy the concept of a reference monitor [2], shown in Figure 2. While these goals are not required for full functionality of USBFILTER, we chose to design for stronger security guarantees to ensure that processes attempting to access hardware USB devices directly would be unable to circumvent our system. We define the specific goals as follows:

- **G1** *Complete Mediation*. All physical or virtual USB packets must pass through USBFILTER before delivery to the intended destination.
- **G2** *Tamperproof*. USBFILTER may not be bypassed or disabled as long as the integrity of the operating system is maintained.
- **G3** *Verifiable*. The user-defined rules input into the system must be verifiably correct. These rules may not conflict with each other.
- **G4** *Granular*. Any mutable data in a USB packet header must be accessible by a user-defined rule. If the ultimate destination of a packet is a userspace process, USBFILTER must permit the user to specify the process in a rule.
- **G5** *Modular*. USBFILTER must be extensible and allow users to provide submodules to support additional types of analysis.

### 3.3 Design and Implementation

The core USBFILTER component is statically compiled and linked into the Linux kernel image, which hooks the

![Diagram of USBFILTER architecture]

**Figure 3**: The architecture of USBFILTER.
Goals

• Complete mediation
• Tamperproof
• Verifiability
• Granularity
• Extensibility

Reference Monitor
## Rule constructions

<table>
<thead>
<tr>
<th>Category</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>pid, ppid, pgid, uid, euid, gid, egid, comm</td>
</tr>
<tr>
<td>Device</td>
<td>bus#, dev#, port#, if#, devpath, manufacturer, product, serial</td>
</tr>
<tr>
<td>Packet</td>
<td>type, direction, endpoint, address</td>
</tr>
<tr>
<td>LUM</td>
<td>name</td>
</tr>
</tbody>
</table>
Rule consistency

- **General conflict**
  \[
  \text{general\_conflict}(R_a, R_b) \leftarrow \forall C_i \in C : \\
  (\exists C_i^a \ni R_a \land \exists C_i^b \ni R_b \land \text{value}(C_i^a) \neq \text{value}(C_i^b)) \lor \\
  (\exists C_i^a \ni R_a \land \neg C_i^b \ni R_b) \lor \\
  (\neg C_i^a \ni R_a \land \neg C_i^b \ni R_b).
  \]

- **Weak conflict**
  \[
  \text{weak\_conflict}(R_a, R_b) \leftarrow \neg \text{general\_conflict}(R_a, R_b) \land \text{action}(R_a) = \text{action}(R_b).
  \]

- **Strong conflict**
  \[
  \text{strong\_conflict}(R_a, R_b) \leftarrow \neg \text{general\_conflict}(R_a, R_b) \land \text{action}(R_a) \neq \text{action}(R_b).
  \]
Linux USBFILTER Module (LUM)

- User-defined extension for USBFILTER
  - `<linux/usbfilter.h>`
- Rule construction unit
  - writing new rules with LUM
- Looking into the USB packet
  - SCSI commands, IP packets, HID packets, and etc.
LUM: detect the SCSI write cmd

```c
int lbsw_filter_urb(struct urb *urb) {
    char opcode;

    /* Has to be an OUT packet */
    if (usb_pipein(urb->pipe))
        return 0;

    /* Make sure the packet is large enough */
    if (urb->transfer_buffer_length <= LUM_SCSI_CMD_IDX)
        return 0;

    /* Make sure the packet is not empty */
    if (!urb->transfer_buffer)
        return 0;

    /* Get the SCSI cmd opcode */
    opcode = ((char *)urb->transfer_buffer)[LUM_SCSI_CMD_IDX];

    /* Current only handle WRITE_10 for Kingston */
    switch (opcode) {
    case WRITE_10:
        return 1;
    default:
        break;
    }

    return 0;
}
```

Figure 9: An example Linux USBFILTER Module that blocks writes to USB removable storage.
Overview

- **USBFILTER** - 27 kernel source files
  - 4 new files, 23 modified files
  - Across USB, SCSI, Block, and Networking subsystems
- **USBTABLES**
  - Internal Prolog engine
  - 21 rule constructions
Stop BadUSB attacks

For my keyboard/mouse:

```bash
usbtables -a mymou```
For Logitech webcam C310:

```
usbtables -a skype -o uid=1001,comm=skype -v serial=B4482A20 -t allow
usbtables -a nowebcam -v serial=B4482A20 -t drop
```
For any USB storage devices:

```
usbtables -a nodataexfil4
  -l name=block_scsi_write -t drop
```
For Logitech USB headset:

```
usbtables -a logitech-headset -v ifnum=2,product="Logitech USB Headset",manufacturer=Logitech -k
direction=1 -t drop
```
For Nexus 4:

```bash
usbtables -a n4-charger -v product="Nexus 4" -t drop
```

For any phone:

```bash
usbtables -a charger -v busnum=1, portnum=4 -t drop
```
### USBTABLES:

<table>
<thead>
<tr>
<th>Adding a new rule</th>
<th>Avg (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 Base Rules</td>
<td>5.9</td>
</tr>
<tr>
<td>100 Base Rules</td>
<td>5.9</td>
</tr>
</tbody>
</table>

### USBFILTER:

<table>
<thead>
<tr>
<th>Packet filtering</th>
<th>Avg (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 Base Rules</td>
<td>2.6</td>
</tr>
<tr>
<td>100 Base Rules</td>
<td>9.7</td>
</tr>
</tbody>
</table>
Figure 5: Filebench throughput (MB/s) using fileserver workload with different mean file sizes.

Figure 6: Iperf bandwidth (MB/s) using TCP with different time intervals.

Figure 7: Iperf bandwidth (MB/s) using UDP with different time intervals.

Figure 8: Performance comparison of real-world workloads.

Rule Adding

<table>
<thead>
<tr>
<th></th>
<th>Min</th>
<th>Avg</th>
<th>Med</th>
<th>Max</th>
<th>Dev</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (20 rules)</td>
<td>5.1</td>
<td>5.9</td>
<td>6.1</td>
<td>6.6</td>
<td>0.3</td>
</tr>
<tr>
<td>Time (100 rules)</td>
<td>4.9</td>
<td>5.9</td>
<td>6.1</td>
<td>6.8</td>
<td>0.4</td>
</tr>
</tbody>
</table>

Table 2: Rule adding operation time (ms) averaged by 100 runs.

We measured 100 trials of each test. The performance of the Prolog engine is shown in Table 1. The average time used by the Prolog engine is 239.8 µs with 20 rules and 251.7 µs with 100 rules. This fast speed is the result of using GNU Prolog (gplc) compiler to compile Prolog into assembly for acceleration. We also measure the overhead for USBTABLES to add a new rule to the kernel space. This includes loading existing rules into the Prolog engine, checking for conflicts, saving the rule locally, passing the rule to the kernel, and waiting for the acknowledgment. As shown in Table 2, the average time of adding a rule using USBTABLES stays at around 6 ms in both cases, which is a negligible one-time cost.

Table 3: USB enumeration time (ms) averaged by 20 runs.

<table>
<thead>
<tr>
<th></th>
<th>Min</th>
<th>Avg</th>
<th>Med</th>
<th>Max</th>
<th>Dev</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stock Kernel</td>
<td>32.0</td>
<td>33.9</td>
<td>34.1</td>
<td>34.8</td>
<td>0.6</td>
<td>N/A</td>
</tr>
<tr>
<td>USBFILTER (20 rules)</td>
<td>33.2</td>
<td>34.4</td>
<td>34.3</td>
<td>35.8</td>
<td>0.7</td>
<td>1.5%</td>
</tr>
<tr>
<td>USBFILTER (100 rules)</td>
<td>33.9</td>
<td>34.8</td>
<td>34.6</td>
<td>36.0</td>
<td>0.5</td>
<td>2.7%</td>
</tr>
</tbody>
</table>

We manually plugged the headset into the host 20 times. We then compare the results between the USBFILTER kernel with varying numbers of rules loaded and the stock Ubuntu kernel, where USBFILTER is fully disabled.

USB Enumeration Overhead. For this test, we used the Logitech H390 USB headset, which has 4 interfaces.
### Latency

<table>
<thead>
<tr>
<th>Latency (µs)</th>
<th>1 KB</th>
<th>10 KB</th>
<th>100 KB</th>
<th>1 MB</th>
<th>10 MB</th>
<th>100 MB</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Stock</strong></td>
<td>97.6</td>
<td>98.1</td>
<td>99.2</td>
<td>105.5</td>
<td>741.7</td>
<td>5177.7</td>
</tr>
<tr>
<td><strong>USBFILTER</strong></td>
<td>97.7</td>
<td>98.2</td>
<td>99.6</td>
<td>106.3</td>
<td>851.5</td>
<td>6088.4</td>
</tr>
<tr>
<td><strong>Overhead</strong></td>
<td>0.1%</td>
<td>0.1%</td>
<td>0.4%</td>
<td>0.8%</td>
<td>14.8%</td>
<td>17.6%</td>
</tr>
</tbody>
</table>
Performance in real world

![Graph showing performance comparison between Stock and usbfilter for different real-world workloads: KVM, Chrome, ClamAV, and wget. The graph indicates that usbfilter generally outperforms Stock in terms of Time/Scores for each workload.]
Limitations & Future Work

- IRQ contexts
- Vendor-specific drivers
- Response-path filtering
- Making it faster - BPF
- More useful LUMs
- Usability - targeting administrators
Conclusion

- **USBFILTER**
  - A USB layer firewall in the Linux kernel
- **USBTABLES**
  - A user-space tool to manage policies/rules
- Controlling USB device behaviors
  - Defending against BadUSB attacks
  - Limiting USB device functionalities
- Introducing minimum overhead
Get USBFILTER now:  
https://github.com/daveti/usbfILTER

All bugs are introduced by:  
root@davejingtian.org

Thanks!
rules, which can be overwritten by new rules. Future work will add general
scope of an existing rule while avoiding the time gap
strong conflicted with both rule A and B, since rule C
unblock the Kingston storage by writing rule C, rule C
same destination and action. When the user wants to
has a different action, and will never work as expected
is generally conflicted with another rule
contradictory with, a sub rule of, or the same as another
not contain all conditions. For example, a rule can be
ward, there are different conflicting cases between the
rule is correct.
range, and it does not conflict with any existing rules, the
existing rules maintained by
However, depending on the requirement and the imple-
duplicate of an existing rule, strong conflict presents
conditions and the action, particularly when a rule does
between two rules as follows:
C
R
8
3
R
6
9
C
b
i
a
R
69
R
C
b
i
a
R
69
a
R
69
R
C
b
i
a
R
69
C
b
i
a
R
69

USBTABLES -h

-d|--debug enable debug mode
-c|--config path to configuration file (TBD)
-h|--help display this help message
-p|--dump dump all the rules
-a|--add add a new rule
-r|--remove remove an existing rule
-s|--sync synchronize rules with kernel
-e|--enable enable usbfilter
-q|--disable disable usbfilter
-b|--behave change the default behavior
-o|--proc process table rule
-v|--dev device table rule
-k|--pkt packet table rule
-l|--lum LUM table rule
-t|--act table rule action

---------------------------------
proc: pid,ppid,pgid,uid,euid,gid,egid,comm
dev: busnum,devnum,portnum,ifnum,devpath,product,
manufacturer,serial
pkt: types,direction,endpoint,address
lum: name
behavior/action: allow|drop
# A LUM written by dtrump

```c
/* lbsw - A LUM kernel module */
/* used to block SCSI write command within USB packets */

#include <linux/module.h>
#include <linux/usbfilter.h>
#include <scsi/scsi.h>

#define LUM_NAME "block_scsi_write"
#define LUM_SCSI_CMD_IDX 15

struct usbfilter_lum lbsw;
static int lum_registered;

int lbsw_filter_urb(struct urb *urb)
{
    char opcode;
    /* Has to be an OUT packet */
    if (usb_pipein(urb->pipe))
    {
        return 0;
    }
    /* Make sure the packet is large enough */
    if (urb->transfer_buffer_length <= LUM_SCSI_CMD_IDX)
    {
        return 0;
    }
    /* Make sure the packet is not empty */
    if (!urb->transfer_buffer)
    {
        return 0;
    }
    /* Get the SCSI cmd opcode */
    opcode = ((char *)urb->transfer_buffer)[LUM_SCSI_CMD_IDX];
    /* Current only handle WRITE_10 for Kingston */
    switch (opcode) {
    case WRITE_10:
        return 1;
    default:
    break;
    }
    return 0;
}

static int __init lbsw_init(void)
{
    pr_info("lbsw: Entering: %s", __func__);
    snprintf(lbsw.name, USBFILTER_LUM_NAME_LEN, "%s", LUM_NAME);
    lbsw.lum_filter_urb = lbsw_filter_urb;

    /* Register this lum */
    if (usbfilter_register_lum(&lbsw))
    {
        pr_err("usbfilter_register_lum(&lbsw) failed\n");
    } else
    {
        lum_registered = 1;
    }
    return 0;
}

static void __exit lbsw_exit(void)
{
    pr_info("exiting lbsw module\n");
    if (lum_registered)
    {
        usbfilter_deregister_lum(&lbsw);
    }
    module_init(lbsw_init);
    module_exit(lbsw_exit);
    MODULE_LICENSE("GPL");
    MODULE_DESCRIPTION("lbsw module");
    MODULE_AUTHOR("dtrump");
}
```
Just read, seriously

For Kingston USB flash drive:

```bash
usbtables -a nodataexfil -v manufacturer=Kingston
   -l name=block_scsi_write -t drop
usbtables -a nodataexfil2 -o uid=1001
   -v manufacturer=Kingston
   -l name=block_scsi_write -t drop
usbtables -a nodataexfil3 -o comm=vim
   -v manufacturer=Kingston
   -l name=block_scsi_write -t drop
```
What is wrong with USB

• Unlimited capabilities
• No authentication
• BadUSB attacks