LBM: A Security Framework for Peripherals within the Linux Kernel

Dave (Jing) Tian*,^ Grant Hernandez*, Joseph Choi*, Vanessa Frost*, Peter Johnson**, and Kevin Butler*

*University of Florida, Gainesville, FL
^Purdue University, West Lafayette, IN
**Middlebury College, VT

May 22, 2019
Peripherals
Modern Peripherals

USB

Bluetooth

NFC
Modern Peripherals

- Up to 40Gb/s data transfer
- Supports up to two 5K displays
- Charge and provide power from any port
- Up to four Thunderbolt 3 ports
Modern Peripherals

What could possibly go wrong??

Florida Institute of Cyber Security (FICS) Research
Malicious Peripherals

Exploiting Decades-Old Telephone Tech to Break Into Android Device

EXPLOITING DECADES-OLD TELEPHONE TECH TO BREAK INTO ANDROID DEVICES
Malicious Peripherals

5.3 Billion Devices Affected

BadBluetooth Attacks

Oday attacks over NFC!

From: Suren Baghdasaryan <surenb@gmail.com>

When handling SMHLC I-Frame commands "pipe" field used for indexing into an array should be checked before usage. If left unchecked it might access memory outside of the array of size NFC_HCI_MAXPIPES(127).

cc: Stable <stable@vger.kernel.org>
Signed-off-by: Suren Baghdasaryan <surenb@gmail.com>
Signed-off-by: Amit Pundir <amit.pundir@linaro.org>
Reviewed-by: Andy Shevchenko <andriy.shevchenko@linux.intel.com>
Solution?

- Functional-unexpected pkt
- Specially-crafted pkt
- Malformed pkt

Firewall
Challenges

• **Peripheral Diversity**
  - USBFILTER (USENIX Sec’16), USBFirewall (ACSAC’17)
  - Bluetooth, NFC, etc.

• **Filtering (Rule) Complexity**
  - Programmability vs. Usability
  - Extensibility
Linux (e)BPF Modules (LBM)

- A generic security framework for peripherals
  - Peripheral agnostic
  - LBM hooks
  - eBPF
  - Filter DSL
  - Module extension
  - USB, Bluetooth, NFC

Peripheral Diversity

Filtering Complexity

```
BPF Instruction 1
BPF Instruction 2
BPF Instruction 3
CALL BPF_Helper_#1
BPF Instruction 5
BPF Instruction 6
CALL BPF_Helper_#2
```

BPF_Helper_#1: Kernel API X; Kernel API Y; return;
BPF_Helper_#2: Kernel API Z; return;

eBPF Program

Kernel Space
LBM: Architecture

- LBM draws inspiration from the state-of-the-art solutions—USBFirewall and LUM (Linux Usbfilter Module)—for programs executed by LBM.
- LBM unifies support for different subsystems, including USBfilter and USBFirewall, providing a superset of their properties.
- LBM achieves complete mediation for different LBM kernel modules (e.g., lbm1-lbm3) that can be plugged and/or reloaded as a kernel module, thus bypassing security mechanisms enforced by the Linux kernel or third-party USB drivers.
- LBM implements packet filtering in the kernel space and executing these programs for peripheral subsystems, which helps reduce the potential impact from vulnerabilities.

**Diagram:**
- **LBM** framework includes:
  - **Lbmtool**, **Lbmtool**, and **Lbmtool** for peripheral subsystems (USB, Bluetooth, NFC).
  - **BPF/eBPF** for packet rewriting (e.g., changing the port number of a TCP packet).
  - **LVM/Clang** for program verification.
- **User-defined Rules** are implemented by supporting all features provided by existing solutions and extending support to other peripherals, such as Bluetooth.
- **USB Packet**, **BT Packet**, and **NFC Packet** are processed by **LBM TX** and **LBM RX** for different peripherals.

**Table I: LBM vs. usbfilter vs. USBFirewall**

<table>
<thead>
<tr>
<th>Support/Solution</th>
<th>USBfilter</th>
<th>USBFirewall</th>
<th>LBM</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Stack Protection</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>- Complete Mediation</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>- Verifiability</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>- Generality</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>- Module Plugin</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>- User-defined Rules</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Notes:**
- To ease support for a new kind of peripheral, we design a unified framework with different LBM kernel modules (e.g., lbm1-lbm3).
- We design LBM as a standalone kernel component/subsys for programs executed by LBM.
- Depending on different subsystems, (e.g., urb for USB and ioreq for NFC), we define a unified API used by different subsystems to hook into LBM.
- Extending support to other peripherals, such as Bluetooth, improves the LBM framework and essentially hooks into the TX and RX paths for different peripherals.
- As we will later discuss in detail, we convert a LUM (Linux Usbfilter Module) into an LBM kernel module.
LBM: Hooks

- **Linux Security Modules (LSM)**
  - > 100 (kernel 4.13)

- **Linux (e)BPF Modules (LBM)**
  - `int lbm_filter_pkt(
      int subsys, int dir, void *pkt)
  `

```c
void lbm_filter_pkt(LBM_SUBSYS_INDEX_USB, LBM_DIR_TX, (void *)urb);
void lbm_filter_pkt(LBM_SUBSYS_INDEX_USB, LBM_DIR_RX, (void *)urb);
```
its driver, which communicates with USB peripherals directly. Enough without relying on the implementation of certain hooks. Second, these hooks should be general to the Linux kernel, we follow two general rules on the placement.

For each kind of peripheral that LBM supports, we need two hooks: one for the RX path and one for the TX path. To solve these challenges, we deploy only one LBM hook at a time.

To maintain the correctness of the hooks, we leverage the eBPF verifier to examine each eBPF program before it can be loaded. Unlike normal eBPF programs (mainly used by the networking subsystem) loaded into the kernel image like USBFirewall, we transform them to convert a LUM (Linux Usbfilter Module) into an LBM program before loading.

Table I: LBM vs. usbfilter vs. USBFirewall. LBM unifies usbfilter and USBFirewall, providing a superset of their properties.
And elaborate it in the following section.
– high performance. As opposed to the CNF-style DSL used
environment for peripherals and executes eBPF programs as
as the filtering mechanism in LBM as a balance between
achieve
crafting eBPF instructions. The design of the LBM framework
can generate a new dialect for the new peripheral based on a
skb for Bluetooth). Once this LBM hook is placed into a

Table II: LBM vs. usbfilter vs. USBFirewall, specifically with

<table>
<thead>
<tr>
<th></th>
<th>LBMTOOL</th>
<th>usbfilter</th>
<th>USBFirewall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acceleration</td>
<td>G5</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>JIT</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Compilation</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>User-space DSL</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To match on a Dell mouse, keyboard, printer, and
write multiple rules, or we could consolidate them into a larger
If we want to include more than one Dell product, we could
USB device's vendor and product ID, such as a Dell optical
A specific subsystem: USB, Bluetooth, or NFC.

With a compiled LBM program,
LBMTOOL
With
-lmbench
LBM
-usb

Intermediate Representation
LBM Program
B. Compiler Example
Figure 11: The compilation stages of an LBM expression.

L4_:
L3_:
L2_:
L1_:
L0_:
L1_:
L2_:
L3_:
L4_:
L5_:
L6_:
L5_:
LEND:

eBPF Assembly
LSTART:
MOV64_REG(REG_4, 0)
MOV64_REG(REG_1, REG_0)
MOV64_IMM(REG_6, 1)
JMP_IMM(JEQ, REG_1, 16700, L1_)
MOV64_IMM(REG_6, 0)

L1_:
MOV64_REG(REG_1, REG_9)
CALL_FUNC(FUNC_lbm_usb_get_idProduct)
MOV64_REG(REG_2, REG_0)
MOV64_IMM(REG_3, 1)
JMP_IMM(JEQ, REG_2, 12304, L2_)
MOV64_IMM(REG_3, 0)

L2_:
JMP_IMM(JEQ, REG_6, 0, L3_)
JMP_IMM(JEQ, REG_3, 0, L3_)
MOV64_IMM(REG_4, 1)
JMP_A(L4_)

L3_:
MOV64_IMM(REG_4, 0)

L4_:
JMP_IMM(JNE, REG_4, 0, L5_)
MOV64_IMM(REG_0, 0)
EXIT_INSN()

L5_:
MOV64_IMM(REG_0, 1)
EXIT_INSN()
**LBM: Proof-of-Concept for NFC**

**Step I: Place hook**

```c
lbm_filter_pkt(LBM_SUBSYS_INDEX_NFC, LBM_DIR_TX, (void *)skb);
lbm_filter_pkt(LBM_SUBSYS_INDEX_NFC, LBM_DIR_RX, (void *)skb);
```

**Step II: Expose protocol fields**

```c
15 +BPF_CALL_1(lbm_nfc_nci_get_mt, struct sk_buff *, skb)
16 +{
17 +    return nci_mt(skb->data);
18 +}
19 +
20 +static const struct bpf_func_proto lbm_nfc_nci_get_mt_proto = {
21 +    .func = lbm_nfc_nci_get_mt,
22 +    .gpl_only = false,
23 +    .ret_type = RET_INTEGER,
24 +    .arg1_type = ARG_PTR_TO_CTX,
25 +};
```

**Step III: Extend lbmtool**

```c
+nfnc_nci = {
  +"len" : SymbolContext(ty=Type.TY_INT_32, offset=0),
  +"mt" : SymbolHelper(ty=Type.TY_INT_32, name="lbm_nfc_nci_get_mt"),
+
}
```

To double-check that LBM introduces a minimal overhead across different kernel configurations.

<table>
<thead>
<tr>
<th>NFC</th>
<th>Kernel</th>
<th>lbmtool</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td># of lines</td>
<td>85</td>
<td>12</td>
<td>97</td>
</tr>
</tbody>
</table>
LBM: FaceDancer Testing
LBM: Protocol Stack Protection

```c
((usb.request[0] == 0x80) && /* Get_Descriptor */
((usb.request[1] == 0x06) &&
/* Make sure response contains at least 2 bytes */
((usb.actual_length < 2) ||
/* Make sure the descriptor type matches */
((usb.request[3] != usb.data[1]) ||
/* Device descriptor */
((usb.request[3] == 1) && ((usb.data[0] != 18) || (usb.actual_length != 18))) ||
/* Configuration descriptor */
((usb.request[3] == 2) && ((usb.data[0] < 9) || (usb.actual_length < 9))) ||
/* String descriptor */
((usb.request[4] == 3) && ((usb.data[0] < 4) || (usb.actual_length < 4))))
```
LBM: USB Security

• Defending against BadUSB

```c
((usb.pipe == 1) && /* INT (Keystroke) */
 ((usb.manufacturer != "X") ||
  (usb.product != "Y") ||
  (usb.serial != "Z") ||
  (usb.plugtime != 12345))
```

• Securing charging

```c
((usb.busnum == 1) && (usb.portnum == 1))
```
LBM: Bluetooth Security

• Defending against BlueBorne

```c
((bt.l2cap.cid == 0x1) && /* L2CAP Signaling */
/* Configuration Response */
(bt.l2cap.sig.cmd.code == 0x5) &&
(bt.l2cap.sig.cmd.len >= 66))
```

• Defending against BleedingBit

```c
((bt.hci.conn == 1) && /* A link exists */
(bt.hci.conn.type == 0x80)) /* BLE link */
```
For Bluetooth testing, we load LBM rules “HCI-1” and “L2CAP-1” into the system. We implement a simple L2CAP client/server protocol based on PyBluez [1] to generate 10K packets on the RX path for the HCI and L2CAP layers, respectively. As shown in the last four rows of Table VI, the average overheads are 2.81 µs for HCI and 2.93 µs for L2CAP. Again, with the help of JIT, we can reduce the overhead to around 1 µs.

Take away: the general overhead introduced by LBM is around 1 µs for most cases.

D. Macro-Benchmark

For USB, we load the rules “USB-1” and “USB-2” and use filebench [50] to measure the throughput of the USB 3.0 external storage device. We chose the “fileserver” workload model with 10K files, 128KB and 1MB mean file sizes, 10 working threads, and 10-min running time. This workload generates roughly 1GB and 10GBs of files, respectively, within the storage device. As shown in Figure 6, all kernel configurations achieve similar throughput during our testing. When the mean file size is 128KB, the total file size (1 GB) can easily fit into the system page cache. Thus, we are able to achieve close to 500 MB/s throughput (faster than the hard drive's maximum speed of 150 MB/s). When the mean file size is 1MB, the total file size (10 GB) cannot completely fit into the page cache, thus resulting in much lower throughput.

For Bluetooth, we load the rules “HCI-1” and “L2CAP-1” and use l2ping [49] to benchmark the Round-Trip-Time (RTT) for 10K pings. As with the USB testing, all kernel configurations achieve similar RTTs of around 5 ms, as shown in Figure 7. Because the overhead of LBM is under 1 µs in general (Section V-C), the overhead contributed to the RTT measurement is negligible.

To double-check that LBM introduces a minimal overhead across the whole system, we use lmbench [55] to benchmark the whole system across different kernel configurations. The complete summary is available in Appendix C. In short, LBM achieves comparable performance with the vanilla kernel.

E. Scalability

To understand the scalability of LBM, we load the rule “USB-3” into the RX path once, 10 times, and 100 times. As in the micro-benchmark, we record 10K USB packets generated by the USB WiFi adapter and compute the overhead of LBM going through these rules for each packet. As shown in Figure 8, while the total overhead increases as the number of rules increases, the average overhead of checking individual rules decreases. The average overhead was 0.83 µs when there was only one rule loaded. It decreased to 0.32 µs when there were 100 rules loaded. Under JIT, the overhead was further reduced to 0.23 µs. This might be the result of increased cache hits from accessing the same rule in a loop. Even for different rules, it is possible to observe this amortization effect, as long as each rule occupies a different cache line. Also, in

![Graph showing throughput and overhead for LBM and LBM-JIT]
LBM: Discussion

• BPF memory write
• LLVM support
• Stateless vs. Stateful policy
• DMA-oriented protocols
Conclusion

- Linux (e)BPF Module
- USB, Bluetooth, NFC
- Effectiveness and Minimum Overhead

https://github.com/fics/lbm
https://davejingtian.org

Thanks!
Malicious Peripherals

What about wireless peripherals?

5.3 Billion Devices Affected

Oday attacks over NFC!

From: Suren Baghdasaryan <surenb@google.com>

When handling SHDLC I-Frame commands "pipe" field used for indexing into an array should be checked before usage. If left unchecked it might access memory outside of the array of size NFC_HCI_MAX_PIPES(127).

cc: Stable <stable@vger.kernel.org>
Signed-off-by: Suren Baghdasaryan <surenb@google.com>
Signed-off-by: Amit Pundir <amit.pundir@linaro.org>
Reviewed-by: Andy Shevchenko <andriy.shevchenko@linux.intel.com>

BlueBorne
You Will Be Hacked Soon By Bluetooth

Bluetooth Hacking Attack

BlueBorne

Google Pay
LBM: Core Framework

• **An eBPF client**
  • LBM filter = eBPF program

• **Load LBM filters**
  • Subsystem / Path

• **Verify LBM filters**
  • Subsystem / No memory write

• **Store/Manage/Run LBM filters**
  • SysFS (/sys/fs/bpf, /sys/kernel/security/lbm)
**LBM: USB**

- **LBM hooks**
  
  ```
  lbm_filter_pkt(LBM_SUBSYS_INDEX_USB, LBM_DIR_TX, (void *)urb);
  ```

- **34 protocol fields**

- **31 BPF helpers**

- **621 LoC**

---

### LBM hooks

- **LBM hooks**
  
  ```
  lbm_filter_pkt(LBM_SUBSYS_INDEX_USB, LBM_DIR_TX, (void *)urb);
  ```

- **34 protocol fields**

- **31 BPF helpers**

- **621 LoC**
LBM: Bluetooth

- **LBM hooks**

  ```c
  lbm_filter_pkt(LBM_SUBSYS_INDEX_BLUETOOTH, LBM_DIR_TX, (void *)skb);
  lbm_filter_pkt(LBM_SUBSYS_INDEX_BLUETOOTH, LBM_DIR_RX, (void *)skb);
  lbm_filter_pkt(LBM_SUBSYS_INDEX_BLUETOOTH_L2CAP, LBM_DIR_TX, (void *)skb);
  lbm_filter_pkt(LBM_SUBSYS_INDEX_BLUETOOTH_L2CAP, LBM_DIR_RX, (void *)skb);
  ```

- **HCI/L2CAP**
  - 30/28 protocol fields
  - 29/27 BPF helpers
  - 683/744 LoC
LBM: Protocol Stack Protection

/* HCI-CMD */
((bt.hci.type == 1) && (bt.hci.len < 3)) ||

/* HCI-ACL */
((bt.hci.type == 2) && (bt.hci.len < 4)) ||

/* HCI-SCO */
((bt.hci.type == 3) && (bt.hci.len < 3)) ||

/* HCI-EVT */
((bt.hci.type == 4) && (bt.hci.len < 2))
usb.idVendor == 0x413c &&
usb.idProduct == 0x3010

Intermediate Representation

0: t1 := call(lbm_usb_get_idVendor)
1: t0 := binop(EQ, t1, 16700)
2: t3 := call(lbm_usb_get_idProduct)
3: t2 := binop(EQ, t3, 12304)
4: t4 := binop(AND, t0, t2)
LBM: Extended BPF (eBPF)

- 64-bit BPF architecture
- BPF helpers
- BPF maps
- BPF verifier
- BPF JIT

What Went Wrong?

- **No Authorization!**
  - Devices are trusted by default
  - Devices can request any functionality

- **No Integrity!**
  - Device firmware can be hacked
  - Firmware modifications are invisible to host

- **No Authentication!**
  - Devices have no trustworthy notion of identity
Q: How do we support all peripherals??

- USBFILTER (USENIX Security’16)
- Bluetooth-FW, NFC-FW, X-FW?

A: Peripheral Agnostic -

- Separation between mechanism and implementation - hooks
- Separation between mechanism and policy - generic packet filter
#2: Hook Placement

Q: Where to place hooks??

- High layer?
- Low layer?
- In between?

A: Reference Monitor Concept -

- Complete mediation
- Tamperproof / Verifiability
#3: Generic Packet Filter

- **Berkeley Packet Filter (BPF)**
- High-performance (IP) packet filtering
- In-kernel virtual machine (RISC)
- Just-In-Time (JIT) compilation
- Backend of tcpdump

Q: What is generic packet filter??

A: BPF for peripherals!
Q: Who writes filtering rules??

- **End users?**
- **Sysadmins?**
- **Developers?**

A: Everyone! -

- Users not enemy (Doh!)
- Peripheral agnostic (Again!)
BadUSB Attacks

Keystrokes

Data
BadUSB Attacks

USB_pkt (Keystrokes)

USB_pkt (Data)
BlueBorne Attacks

switch (result) {
    case L2CAP_CONF_SUCCESS:
        l2cap_conf_rfc_get(chan, rsp->data, len);
        clear_bit(CONF_REM_CONF_PEND, &chan->conf_state);
        break;

    case L2CAP_CONF_PENDING:
        set_bit(CONF_REM_CONF_PEND, &chan->conf_state);

        if (test_bit(CONF_LOC_CONF_PEND, &chan->conf_state)) {
            char buf[64];

            len = l2cap_parse_conf_rsp(chan, rsp->data, buf, &result);
            if (len < 0) {
                l2cap_send_disconn_req(chan, ECONNRESET);
                goto done;
            }
        }
}

buf length (64) is NOT passed here!