XXX
Tolerating Malicious Drivers in Linux

Silas Boyd-Wickizer and Nickolai Zeldovich
How could a device driver be malicious?

Today's device drivers are highly privileged

  Write kernel memory, allocate memory, ...

Drivers are complex; developers write buggy code

Result: Attackers exploit vulnerabilities
How could a device driver be malicious?

Today's device drivers are highly privileged
Write kernel memory, allocate memory, ...

Drivers are complex; developers write buggy code

Surge of killer device drivers leave no OS safe
How could a device driver be malicious?

Today's device drivers are highly privileged

Write kernel memory, allocate memory...

Drivers are complex; developers write buggy code

Result: Attackers exploit vulnerabilities
Current approach

User-space drivers in \(\mu\)kernels (Minix, L4, ...)
Write device driver in new language (Termite)
Handle common faults (Nooks, microdrivers, ...)
Goal

Secure, efficient, & unmodified drivers on Linux
Previous user-space drivers

μkernel

Hardware

User

Kernel

Ethernet driver

Kernel core

Network stack

Application

User
Previous user-space drivers

- Hardware
- User
- Kernel
- Kernel core
- Ethernet driver
- Network stack
- Application

Confine driver in a process
Previous user-space drivers

Confine driver in a process

General purpose syscall API to configure device
Previous user-space drivers

Confine driver in a process

Confine device with IO virtualization HW.

General purpose syscall API to configure device
Previous user-space drivers

Confine driver in a process

Confine device with IO virtualization HW.

IPC network driver API
E.g. tx_packet

General purpose syscall API to configure device
Current Linux driver architecture
Current Linux driver architecture

Kernel

User

Application

Network stack

Hardware

Kernel runtime (e.g. kmalloc)

Ethernet driver

Kernel RT

netdevice
Current Linux driver architecture

Kernel runtime (e.g. kmalloc)

Network driver API (e.g. tx_packet)

Kernel RT

netdevice

Hardware

Kernel

Ethernet driver

Network stack

Application
Linux user-space driver problem

Kernel RT and driver APIs won't work for untrusted drivers in a different AS
SUD's approach
SUD's approach

SUD UML handles calls to kernel RT
SUD's approach

SUD UML handles calls to kernel RT
Proxy driver and SUD UML allow reuse of existing driver APIs
SUD's approach

SUD UML handles calls to kernel RT
Proxy driver and SUD UML allow reuse of existing driver APIs
SUD's approach

SUD UML handles calls to kernel RT
Proxy driver and SUD UML allow reuse of existing driver APIs
SUD's approach

SUD UML handles calls to kernel RT

Proxy driver and SUD UML allow reuse of existing driver APIs
SUD's results

Tolerate malicious device drivers
Proxy drivers small (~500 LOC)
One proxy driver per device class
Few kernel modifications (~50 LOC)
Unmodified drivers (6 test drivers)
High performance, low overhead

No need for new OS or language
Security challenge: prevent attacks

Problem: driver must perform privileged operations
  Memory access, driver API, DMA, interrupts, ...

Attacks from driver code:
  Direct system attacks: memory corruption, ...
  Driver API attacks: invalid return value, deadlock, ...

Attacks from device:
  DMA to DRAM, peer-to-peer attacks, interrupt storms
Practical challenges

High performance, low overhead

Challenge: interact with hardware and kernel at high rate, kernel-user switch expensive
E.g. Ethernet driver \(~100k\) times a second

Reuse existing drivers and kernel

Challenge: drivers assume fully-privileged kernel env.
Challenge: kernel driver API complex, non-uniform
SUD overview

- Hardware
- User
  - Driver
  - SUD UML
- User
  - Application
- Kernel
  - HW access module
  - Proxy driver
  - Kernel core
Linux driver APIs

Linux defines a driver API for each device class
Driver and kernel functions and variables
Example: wireless driver API

Linux defines a driver API for each device class

Driver and kernel functions and variables

```c
struct wireless_ops {
    int (*tx)(struct sk_buff*);
    int (*configure_filter)(int);
    ...
};

struct wireless_hw {
    int conf;
    int flags
    ....
};
```
Example: wireless driver API

Linux defines a driver API for each device class

Driver and kernel functions and variables

```c
struct wireless_ops {
    int (*tx)(struct sk_buff*);
    int (*configure_filter)(int);
    ...
};

struct wireless_hw {
    int conf;
    int flags
    ....
};
```

Proxy drivers and SUD-UML convert API to RPCs
Example: wireless driver API

Linux defines a driver API for each device class
Driver and kernel functions and variables

```c
struct wireless_ops {
    int (*tx)(struct sk_buff*);
    int (*configure_filter)(int);
    ...
};

struct wireless_hw {
    int conf;
    int flags
    ....
};
```

Proxy drivers and SUD-UML convert API to RPCs
Example: wireless driver API

Linux defines a driver API for each device class

Driver and kernel functions and variables

```c
struct wireless_ops {
    int (*tx)(struct sk_buff*);
    int (*configure_filter)(int);
    ...
};

struct wireless_hw {
    int conf;
    int flags
    ....
};
```

Proxy drivers and SUD-UML convert API to RPCs

Called in a non-preemptable context
Example: wireless driver API

Linux defines a driver API for each device class

Driver and kernel functions and variables

```c
struct wireless_ops {
    int (*tx)(struct sk_buff*);
    int (*configure_filter)(int);
    ...
};

struct wireless_hw {
    int conf;
    int flags
    ....
};
```

Proxy drivers and SUD-UML convert API to RPCs

Called in a non-preemptable context

Driver API variable
Wireless driver in SUD

Basic driver API → SUD RPC API → driver API
Non-preemptable function: implement in proxy
Driver API variable: shadow variables
Example 1: transmit a packet
Example 1: transmit a packet

Hardware

Kernel

User

Wireless proxy driver

Wireless driver

SUD UML

User

Web browser

Wireless core

Socket write
Example 1: transmit a packet
Example 1: transmit a packet
Example 1: transmit a packet
Example 1: transmit a packet
Example 2: non-preemptable callback

Problem: unable to switch to user-space
Example 2: non-preemptable callback

Problem: unable to switch to user-space
Example 2: non-preemptable callback

Problem: unable to switch to user-space
Example 2: non-preemptable callback

Problem: unable to switch to user-space
Example 2: non-preemptable callback

Problem: unable to switch to user-space
Example 2: non-preemptable callback

Problem: unable to switch to user-space
Solution: implement directly in proxy driver
Example 2: non-preemptable callback

Problem: unable to switch to user-space
Solution: implement directly in proxy driver
Example 2: non-preemptable callback

Problem: unable to switch to user-space
Solution: implement directly in proxy driver
Example 2: non-preemptable callback

Problem: unable to switch to user-space
Solution: implement directly in proxy driver

wireless_ops.configure_filter
Example 2: non-preemptable callback

Problem: unable to switch to user-space
Solution: implement directly in proxy driver
Example 3: driver API variables

Problem: user-space can't access API variables
Example 3: driver API variables

Problem: user-space can't access API variables

Diagram showing the relationship between hardware, kernel, user space, and the wireless driver and API variables.
Example 3: driver API variables

Problem: user-space can't access API variables
Example 3: driver API variables

Problem: user-space can't access API variables
Example 3: driver API variables

Problem: user-space can't access API variables

Solution: allocate a shadow copy and synchronize before and after RPCs
Example 3: driver API variables

Problem: user-space can't access API variables

Solution: allocate a shadow copy and synchronize before and after RPCs
Example 3: driver API variables

Problem: user-space can't access API variables

Solution: allocate a shadow copy and synchronize before and after RPCs
Example 3: driver API variables

Problem: user-space can't access API variables

Solution: allocate a shadow copy and synchronize before and after RPCs

- Kernel
- User
- Wireless proxy driver
- Wireless core
- Web browser
- Hardware
- SUD UML
- wireless_hw
- wireless_hw

Synchronize before sending RPC
Example 3: driver API variables

Problem: user-space can't access API variables

Solution: allocate a shadow copy and synchronize before and after RPCs
Example 3: driver API variables

Problem: user-space can't access API variables

Solution: allocate a shadow copy and synchronize before and after RPCs

Diagram:

- Hardware
- Kernel
- User

Wireless driver
SUD UML
wireless_hw

Wireless proxy driver
wireless_hw

Web browser

Reads updates from shadow variable
SUD overview
SUD overview

- Hardware
- User
- Kernel

- Driver
- SUD UML
- Proxy driver
- Kernel core
- Application

HW access module
Attacks from hardware

- CPU
- PCI bus
- DRAM
- Memory interconnect
Attacks from hardware

Driver configures the device to execute attacks
Attacks from hardware

Driver configures the device to execute attacks

DMA to DRAM
Attacks from hardware

Driver configures the device to execute attacks

DMA to DRAM

Peer-to-peer messages
Attacks from hardware

Driver configures the device to execute attacks

- DMA to DRAM
- Peer-to-peer messages
- Interrupt storms
Attacks from hardware

Driver configures the device to execute attacks
  DMA to DRAM
  Peer-to-peer messages
  Interrupt storms

HW access module prevents attacks
  Interposes on driver-device communication
  Uses IO virtualization to provide direct device access
IO virtualization hardware

Diagram:
- APIC interconnect
- CPU
- MSI
- Memory interconnect
- DRAM
- IOMMU
- PCI express switch
IO virtualization hardware

Use IOMMU to map DMA buffer pools
Prevents DMA to DRAM attacks
IO virtualization hardware

Use PCI ACS to prevent peer-to-peer messaging
Prevents peer-to-peer attacks
IO virtualization hardware

Use MSI to mask interrupts
Prevents interrupt storms
Interrupt handlers in Linux
Interrupt handlers in Linux
Interrupt handlers in Linux

Driver called with IRQs disabled (non-preemptable)
Interrupt handlers in Linux

Kernel calls driver interrupt handler
Driver clears interrupt flag
Interrupt handlers with SUD
Interrupt handlers with SUD

Kernel calls HW access module interrupt handler
HW access module masks interrupt with MSI
Interrupt handlers with SUD

Kernel calls HW access module interrupt handler
HW access module masks interrupt with MSI
Interrupt handlers with SUD

Kernel calls HW access module interrupt handler
HW access module masks interrupt with MSI
Asynchronous RPC to driver
Interrupt handlers with SUD

Kernel calls HW access module interrupt handler
HW access module masks interrupt with MSI
Asynchronous RPC to driver
Driver clears interrupt
Interrupt handlers with SUD

HW access module masks interrupt with MSI
Asynchronous RPC to driver
Driver clears interrupt
HW access module unmasks MSI
SUD overview

Hardware

Kernel

User

Driver

SUD UML

Proxy driver

Kernel core

Application

HW access module
Prototype of SUD

<table>
<thead>
<tr>
<th>Trusted code</th>
<th>Lines of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCI access module</td>
<td>2800</td>
</tr>
<tr>
<td>Ethernet proxy driver</td>
<td>300</td>
</tr>
<tr>
<td>Wireless proxy driver</td>
<td>600</td>
</tr>
<tr>
<td>Audio proxy driver</td>
<td>550</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Untrusted code</th>
<th>Lines of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>User-mode runtime</td>
<td>5000</td>
</tr>
<tr>
<td>Drivers</td>
<td>5000 – 50,000 (each)</td>
</tr>
</tbody>
</table>

Supports all Ethernet, wireless, USB, audio drivers
Tested: e1000e, ne2k-pci, iwldgn, snd_hda_intel, ehci_hcd, uhci_hcd, ...
## Prototype of SUD

<table>
<thead>
<tr>
<th>Trusted code</th>
<th>Lines of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCI access module</td>
<td>2800</td>
</tr>
<tr>
<td>Ethernet proxy driver</td>
<td>300</td>
</tr>
<tr>
<td>Wireless proxy driver</td>
<td>600</td>
</tr>
<tr>
<td>Audio proxy driver</td>
<td>550</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Untrusted code</th>
<th>Lines of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>User-mode runtime</td>
<td>5000</td>
</tr>
<tr>
<td>Drivers</td>
<td>5000 – 50,000 (each)</td>
</tr>
</tbody>
</table>

Supports all Ethernet, wireless, USB, audio drivers
Tested: e1000e, ne2k-pci, iwlagrn, snd_hda_intel, ehci_hcd, uhci_hcd, ...
## Prototype of SUD

<table>
<thead>
<tr>
<th>Trusted code</th>
<th>Lines of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCI access module</td>
<td>2800</td>
</tr>
<tr>
<td>Ethernet proxy driver</td>
<td>300</td>
</tr>
<tr>
<td>Wireless proxy driver</td>
<td>600</td>
</tr>
<tr>
<td>Audio proxy driver</td>
<td>550</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Untrusted code</th>
<th>Lines of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>User-mode runtime</td>
<td>5000</td>
</tr>
<tr>
<td>Drivers</td>
<td>5000 – 50,000 (each)</td>
</tr>
</tbody>
</table>

Supports all Ethernet, wireless, USB, audio drivers
Tested: e1000e, ne2k-pci, iwlahn, snd_hda_intel, ehci_hcd, uhci_hcd, ...
Performance

For most devices, does not matter
  Printers, cameras, ...

Stress-test: e1000e gigabit network card
  Requires high throughput
  Requires low latency
  Many device driver interactions

Test machine: 1.4GHz dual core Thinkpad
Performance questions?

What performance does SUD get?
   Network throughput, latency

How much does it cost?
   CPU cycles
SUD achieves same device performance

Normalized throughput relative to Linux
TCP: streaming (950 Mbps in both cases)
UDP: one-byte-data packets
CPU cost is low

SUD overhead: user-kernel switch, TLB misses
Overheads not significant for many workloads
(packets larger than min. packet size)
Future directions

Explore hierarchical untrusted device drivers
  PCI bus → SATA controller → SATA disk → …

Explore giving apps direct hardware access
  Safe HW access for network analyzer, X server, …

Performance analysis and optimizations
  SUD specific device drivers, super pages, …
Related work

Mircokernels (Minix, L4, ...)
    Simple drivers, driver API designed for user-space

Nooks, microdrivers
    Handles common bugs, many changes to kernel

Languages (e.g. Termite), source code analysis
    Complimentary to user-space drivers

*No need for new OS or language*
Summary

Driver bugs lead to system crashes or exploits

SUD protects Linux from malicious drivers using proxy drivers and IO virtualization HW

- Runs unmodified Linux device drivers
- High performance, low overheads
- Few modifications to Linux kernel
Security evaluation

Manually constructed potential attacks

  Memory corruption, arbitrary upcall responses, not responding at all, arbitrary DMA, ...

Relied on security heavily during development

  SUD caught all bugs in user-mode driver framework
  No crashes / reboots required to develop drivers

Ideal, but not done: red-team evaluation?