The Role of the Operating System in the Era of Cyberwar

Vasileios (Vasilis) Kemerlis

Department of Computer Science
Brown University

CSCI 1800: Cybersecurity and International Relations, 04/06/2016
The Role of the Kernel

What is a kernel anyway?

"Heart" of the Operating System

- Abstracts the hardware
  - CPU → Execution thread
  - MEM → Virtual address space
  - DEV → I/O object
- Manages resources
  - Performance
  - Protection

The security of a computer system can only be as good as that of the underlying OS kernel!
The Role of the Kernel

What is a kernel anyway?

“Heart” of the Operating System

- Abstracts the hardware
  - CPU → Execution thread
  - MEM → Virtual address space
  - DEV → I/O object

- Manages resources
  - Performance
  - Protection
    - Isolation → Virtual memory
    - Confinement → Namespaces, Capabilities
    - Access control → DAC/MAC, ACLs

The security of a computer system can only be as good as that of the underlying OS kernel!
The Role of the Kernel

What is a kernel anyway?

“Heart” of the Operating System

- Abstracts the hardware
  - CPU $\leadsto$ Execution thread
  - MEM $\leadsto$ Virtual address space
  - DEV $\leadsto$ I/O object

- Manages resources
  - Performance
  - Protection
    - Isolation $\leadsto$ Virtual memory
    - Confinement $\leadsto$ Namespaces, Capabilities
    - Access control $\leadsto$ DAC/MAC, ACLs

Kernel $\rightarrow$ Privilege

*The security of a computer system can only be as good as that of the underlying OS kernel!*
The Kernel as a Target

Why care?

Increased focus on kernel exploitation

- iOS kernel exploits used for jailbreaking (v3.x – v9.x)
  - IOHIDFamily (CVE-2015-6974)
  - IOSharedDataQueue (CVE-2014-4461)
  - TempSensor (CVE-2014-4388)
  - ptmx_get_ioctl (CVE-2014-1278)
  - IOUSBDeviceFamily (CVE-2013-0981)
  - Debug syscall (CVE-2012-0643)
  - Packet Filter (CVE-2012-3728)
  - HFS Heap (CVE-2012-0642)
  - ndrv_setspec
  - HFS Legacy Volume Name
  - Packet Filter (CVE-2010-3830)
  - IOSurface (CVE-2010-2973)
  - BPF_STX
The Kernel as a Target

Why care?

Increased focus on kernel exploitation

- iOS kernel exploits used for jailbreaking (v3.x – v9.x)
  - IOHIDFamily (CVE-2015-6974)
  - IOSharedDataQueue (CVE-2014-4461)
  - TempSensor (CVE-2014-4388)
  - ptmx_get_ioctl (CVE-2014-1278)
  - IOUSBDeviceFamily (CVE-2013-0981)
  - Debug syscall (CVE-2012-0643)
  - Packet Filter (CVE-2012-3728)
  - HFS Heap (CVE-2012-0642)
  - ndrv_setspec
  - HFS Legacy Volume Name
  - Packet Filter (CVE-2010-3830)
  - IOSurface (CVE-2010-2973)
  - BPF_STX
The Kernel as a Target

Why care?

Increased focus on kernel exploitation

- Linux kernel exploits used by Android malware
  - TowelRoot (CVE-2014-3153)
  - perf_swevent_init (CVE-2013-2094)
  - Levitator (CVE-2011-1350)
  - Wunderbar (CVE-2009-2692)
  - ...

TowelRoot (CVE-2014-3153)
perf_swevent_init (CVE-2013-2094)
Levitator (CVE-2011-1350)
Wunderbar (CVE-2009-2692)
...
The Kernel as a Target

Why care?

Increased focus on kernel exploitation

- Windows kernel exploits used in cyber attacks
  - Duqu2 (CVE-2015-2360)
  - Pawn Storm (CVE-2015-1701)
  - Sandworm (CVE-2014-4114)
  - TrueType Font Parsing (CVE-2011-3042)
  - ...

[Map showing known targets with NATO, Ukraine, France, and Poland connected by arrows labeled Sand Worm Team.]
The Kernel as a Target

Why care?

Increased focus on kernel exploitation

1. High-value asset → **Privileged** piece of code
   - Responsible for the integrity of OS security mechanisms

2. Large attack surface → syscalls, device drivers, pseudo fs, ...
   - New features & optimizations → **New attack opportunities**

3. Exploiting privileged userland processes has become harder → Canaries+ASLR+W^X+Fortify+RELRO+BIND_NOW+BPF_SECCOMP+...
   - Sergey Glazunov (Pwn2Own ’12) ~→ 14 bugs to takedown Chrome

“A Tale of Two Pwnies” (http://blog.chromium.org)
Kernel Vulnerabilities

Current state of affairs (all vendors)

- Kernel stack smashing
- Kernel heap overflows
- Wild writes, off-by-\( n \)
- Poor arg. sanitization
- User-after-free, double free, dangling pointers
- Signedness errors, integer overflows
- Race conditions, memory leaks
- Missing authorization checks

Source: National Vulnerability Database (http://nvd.nist.gov)

vpk@cs.brown.edu (Brown University)
Kernel Vulnerabilities (cont’d)

Current state of affairs (Linux only)

Linux kernel vulnerabilities per year

<table>
<thead>
<tr>
<th>Year</th>
<th># of vulnerabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>1999</td>
<td>10</td>
</tr>
<tr>
<td>2000</td>
<td>15</td>
</tr>
<tr>
<td>2001</td>
<td>17</td>
</tr>
<tr>
<td>2002</td>
<td>21</td>
</tr>
<tr>
<td>2003</td>
<td>23</td>
</tr>
<tr>
<td>2004</td>
<td>34</td>
</tr>
<tr>
<td>2005</td>
<td>94</td>
</tr>
<tr>
<td>2006</td>
<td>136</td>
</tr>
<tr>
<td>2007</td>
<td>112</td>
</tr>
<tr>
<td>2008</td>
<td>86</td>
</tr>
<tr>
<td>2009</td>
<td>92</td>
</tr>
<tr>
<td>2010</td>
<td>103</td>
</tr>
<tr>
<td>2011</td>
<td>90</td>
</tr>
<tr>
<td>2012</td>
<td>86</td>
</tr>
<tr>
<td>2013</td>
<td>180</td>
</tr>
<tr>
<td>2014</td>
<td>165</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Year</th>
<th># of vulnerabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>2005</td>
<td>94</td>
</tr>
<tr>
<td>2006</td>
<td>136</td>
</tr>
<tr>
<td>2007</td>
<td>112</td>
</tr>
<tr>
<td>2008</td>
<td>86</td>
</tr>
<tr>
<td>2009</td>
<td>92</td>
</tr>
<tr>
<td>2010</td>
<td>103</td>
</tr>
<tr>
<td>2011</td>
<td>90</td>
</tr>
<tr>
<td>2012</td>
<td>86</td>
</tr>
<tr>
<td>2013</td>
<td>180</td>
</tr>
<tr>
<td>2014</td>
<td>165</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Year</th>
<th># of vulnerabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>2005</td>
<td>94</td>
</tr>
<tr>
<td>2006</td>
<td>136</td>
</tr>
<tr>
<td>2007</td>
<td>112</td>
</tr>
<tr>
<td>2008</td>
<td>86</td>
</tr>
<tr>
<td>2009</td>
<td>92</td>
</tr>
<tr>
<td>2010</td>
<td>103</td>
</tr>
<tr>
<td>2011</td>
<td>90</td>
</tr>
<tr>
<td>2012</td>
<td>86</td>
</tr>
<tr>
<td>2013</td>
<td>180</td>
</tr>
<tr>
<td>2014</td>
<td>165</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Kernel ver.</th>
<th>Size</th>
<th>Dev. days</th>
<th>Patches</th>
<th>Changes/hr</th>
<th>Fixes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6.11 (03/02/05)</td>
<td>6.6 MLOC</td>
<td>69</td>
<td>3.6K</td>
<td>2.18</td>
<td>79</td>
</tr>
<tr>
<td>3.10 (30/06/13)</td>
<td>16.9 MLOC</td>
<td>63</td>
<td>13.3K</td>
<td>9.02</td>
<td>670</td>
</tr>
</tbody>
</table>

Source: CVE Details (http://www.cvedetails.com), The Linux Foundation
### Kernel Vulnerabilities (cont’d)

**Current state of affairs (Linux only)**

![Linux kernel vulnerabilities per year](image)

<table>
<thead>
<tr>
<th>Year</th>
<th># of vulnerabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>1999</td>
<td>19</td>
</tr>
<tr>
<td>2000</td>
<td>18</td>
</tr>
<tr>
<td>2001</td>
<td>20</td>
</tr>
<tr>
<td>2002</td>
<td>20</td>
</tr>
<tr>
<td>2003</td>
<td>45</td>
</tr>
<tr>
<td>2004</td>
<td>70</td>
</tr>
<tr>
<td>2005</td>
<td>130</td>
</tr>
<tr>
<td>2006</td>
<td>150</td>
</tr>
<tr>
<td>2007</td>
<td>120</td>
</tr>
<tr>
<td>2008</td>
<td>100</td>
</tr>
<tr>
<td>2009</td>
<td>110</td>
</tr>
<tr>
<td>2010</td>
<td>120</td>
</tr>
<tr>
<td>2011</td>
<td>110</td>
</tr>
<tr>
<td>2012</td>
<td>100</td>
</tr>
<tr>
<td>2013</td>
<td>150</td>
</tr>
<tr>
<td>2014</td>
<td>170</td>
</tr>
</tbody>
</table>

**Source:** CVE Details ([http://www.cvedetails.com](http://www.cvedetails.com)), The Linux Foundation

<table>
<thead>
<tr>
<th>Kernel ver.</th>
<th>Size</th>
<th>Dev. days</th>
<th>Patches</th>
<th>Changes/hr</th>
<th>Fixes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6.11 (03/02/05)</td>
<td>6.6 MLOC</td>
<td>69</td>
<td>3.6K</td>
<td>2.18</td>
<td>79</td>
</tr>
<tr>
<td>3.10 (30/06/13)</td>
<td>16.9 MLOC</td>
<td>63</td>
<td>13.3K</td>
<td>9.02</td>
<td>670</td>
</tr>
</tbody>
</table>

Source: CVE Details ([http://www.cvedetails.com](http://www.cvedetails.com)), The Linux Foundation
Kernel Vulnerabilities (cont’d)

Pricelist

**ZERODIUM Payout Ranges**

- LPE: Local Privilege Escalation
- MTB: Mitigation Bypass
- RCE: Remote Code Execution
- RJB: Remote Jailbreak
- SBX: Sandbox Escape
- VME: Virtual Machine Escape

*Source: Zerodium*
Attacking the “Core”

Threats classification

1. **Privilege escalation**
   - Arbitrary code execution $\leadsto$ return-to-user (ret2usr)
   - Kernel stack smashing
   - Kernel heap overflows
   - Wild writes, off-by-n
   - Poor arg. sanitization

2. **Persistent foothold**
   - Kernel object hooking (KOH) $\leadsto$ control-flow hijacking
     - Kernel control data (function ptr., dispatch tbl., return addr.)
     - Kernel code (.text)
   - Direct kernel object manipulation (DKOM) $\leadsto$ cloaking
     - Kernel non-control data
Attacking the “Core”

Threats classification

1. **Privilege escalation**
   - Arbitrary code execution \(\rightsquigarrow\) return-to-user (ret2usr)
   - Kernel stack smashing
   - Kernel heap overflows
   - Wild writes, off-by-\(n\)
   - Poor arg. sanitization

2. **Persistent foothold**
   - Kernel object hooking (KOH) \(\rightsquigarrow\) control-flow hijacking
     - Kernel control data (function ptr., dispatch tbl., return addr.)
     - Kernel code (.text)
   - Direct kernel object manipulation (DKOM) \(\rightsquigarrow\) cloaking
     - Kernel non-control data
Attacking the “Core” (cont’d)

From RING 0 to UID 0

- Privilege escalation

![Diagram showing privilege escalation from RING 0 to UID 0]
Attacking the “Core” (cont’d)

From RING 0 to UID 0

- Privilege escalation

![Diagram showing privilege escalation from RING 0 to UID 0 with the kernel and system components highlighted.](image-url)
Attacking the “Core” (cont’d)

From RING 0 to UID 0

- Privilege escalation

```
APP  APP  APP  APP  APP

KERNEL

CPU  MEM  DEV
```
Attacking the “Core” (cont’d)

From RING 0 to UID 0

- Privilege escalation

![Diagram of privilege escalation]
Attacking the “Core” (cont’d)

From RING 0 to UID 0

- Privilege escalation

![Diagram showing privilege escalation from APP to CPU, MEM, and DEV, ultimately leading to KERNEL. Each step is marked with "PWNED!".]
Code-\{injection, reuse\} Attacks

Linux example (x86)
Introduction
Kernel attacks

Code-\{injection, reuse\} Attacks (cont’d)

Classic defenses

Similar to userland exploitation → Many protection schemes

✓ stack canaries (SSP), SLAB red zones, KASLR, W^X
✓ const dispatch tables (IDT, GDT, syscall)
✓ .rodata sections
✓ ...

[Diagram showing memory layout and protection schemes]
Return-to-user \((\text{ret2usr})\) Attacks

What are they?

Attacks against OS kernels with shared kernel/user address space

- Overwrite kernel code (or data) pointers with \textit{user space} addresses
  - $\times$ return addr., dispatch tbl., function ptr.,
  - $\times$ data ptr.

- Payload $\rightsquigarrow$ Shellcode, ROP payload, tampered-with data structure(s)
  - Placed in user space
  - $\times$ Executed (referenced) in \textit{kernel} context
Return-to-user (\textit{ret2usr}) Attacks

What are they?

Attacks against OS kernels with shared kernel/user address space

- Overwrite kernel code (or data) pointers with \textit{user space} addresses
  - $\times$ return addr., dispatch tbl., function ptr.,
  - $\times$ data ptr.
- Payload $\leadsto$ Shellcode, ROP payload, tampered-with data structure(s)
  - $\blacktriangleright$ Placed in user space
  - $\times$ Executed (referenced) in kernel context
- De facto kernel exploitation technique
  - $\times$ http://www.exploit-db.com/exploits/34134/ (21/07/14)
  - $\times$ http://www.exploit-db.com/exploits/131/ (05/12/03)

vpk@cs.brown.edu (Brown University)
ret2usr Attacks (cont’d)

Retro pwn!

**ret2usr Attacks (cont’d)**

Retro pwn!


**PDP-10 address wraparound fault**

“Control was therefore returned to user code at his virtual location zero—in privileged mode!”

ret2usr Attacks (cont’d)

Why do they work?

Weak address space (kernel/user) separation

- Shared kernel/process model → **Performance**
  - \( \text{cost}(\text{mode\_switch}) \ll \text{cost}(\text{context\_switch}) \)
- The kernel is protected from userland → Hardware-assisted isolation
  - The opposite is **not** true
  - Kernel \( \rightsquigarrow \) **ambient authority** (unrestricted access to all memory and system objects)
ret2usr Attacks (cont’d)

Why do they work?

Weak address space (kernel/user) separation

- Shared kernel/process model → Performance
  - ✓ $\text{cost(mode\_switch)} \ll \text{cost(context\_switch)}$

- The kernel is protected from userland → Hardware-assisted isolation
  - ✗ The opposite is not true
  - ✗ Kernel $\rightsquigarrow$ ambient authority (unrestricted access to all memory and system objects)

- The attacker completely controls user space memory
  - ● Contents & perms.
kGuard [USENIX Sec '12, ;login: '12]
Lightweight protection against ret2usr

- **Cross-platform** solution that enforces address space separation between user and kernel space
  - x86, x86-64, ARM, ...
  - Linux, Android, {Free, Open}BSD, ...
- **Defensive mechanism** that builds upon inline monitoring and code diversification
- **Non-intrusive & low overhead**

**Goal**

 ✓ **Cast a realistic** threat ineffective
kGuard Design
Control-flow assertions (key technology #1; confinement)

- Compact, inline guards injected at compile time
  - Two flavors \( \sim \) CFA\(_R\) & CFA\(_M\)
- Placed before every exploitable control-flow transfer
  - `call`, `jmp`, `ret` in x86/x86-64
  - `ldm`, `blx`, ..., in ARM
**kGuard Design**

Control-flow assertions (key technology #1; confinement)

- Compact, inline guards injected at compile time
  - Two flavors \( \rightsquigarrow \text{CFA}_R \& \text{CFA}_M \)
- Placed before every exploitable control-flow transfer
  - `call`, `jmp`, `ret` in x86/x86-64
  - `ldm`, `blx`, ..., in ARM

- Verify that the target address of an **indirect** branch is always inside kernel space

- If the assertion is true, execution continues normally; otherwise, control-flow is transferred to a runtime violation handler
kGuard Design (cont’d)

CFA example

Indirect call in drivers/cpufreq/cpufreq.c (x86 Linux)
kGuard Design (cont’d)

CFA example

```
push %edi
  lea 0x50(%ebx),%edi
  cmp $0xc0000000,%edi
  jae lbl1
  pop %edi
  call 0xc05af8f1

lbl1:
  pop %edi
  cmpl $0xc0000000,0x50(%ebx)
  jae lbl2
  movl $0xc05af8f1,0x50(%ebx)

lbl2:
  call *0x50(%ebx)
```

Indirect call in `net/socket.c` (x86 Linux)
kGuard Design (cont’d)

CFA\textsubscript{M} example

```assembly
push %edi
lea 0x50(%ebx),%edi
cmp $0xc0000000,%edi
jae lbl1
pop %edi
call 0xc05af8f1

lbl1: pop %edi
cmpl $0xc0000000,0x50(%ebx)
jae lbl2
movl $0xc05af8f1,0x50(%ebx)

lbl2: call *0x50(%ebx)
```

Indirect call in `net/socket.c` (x86 Linux)
Bypassing kGuard
Bypass trampolines

- CFAs provide reliable protection if the attacker partially controls a computed branch target
- What about vulnerabilities that allow overwriting kernel memory with arbitrary values?
Bypassing kGuard

Bypass trampolines

- CFAs provide reliable protection if the attacker **partially** controls a computed branch target
- What about vulnerabilities that allow overwriting kernel memory with **arbitrary** values?

**Attacking kGuard**

1. Find **two** computed branch instructions whose operands can be reliably overwritten
2. Overwrite the value (branch target) of the first with the address of the second
3. Overwrite the value of the second with a user-space address
**Countermeasures**

**Code inflation (key technology #2; diversification)**

- **Reshape kernel’s `.text` area**
  - Insert a random NOP sled at the ***beginning*** of `.text`
  - Inject a NOP sled of random length ***before*** every CFA

  - Each NOP sled “pushes” further instructions at higher memory addresses (cumulative effect)
Countermeasures

Code inflation (key technology #2; diversification)

- Reshape kernel’s .text area
  - Insert a random NOP sled at the beginning of .text
  - Inject a NOP sled of random length before every CFA
- Each NOP sled “pushes” further instructions at higher memory addresses (cumulative effect)

Result

- The location of each indirect control transfer is randomized (per build)

Important assumption

- Kernel .text & symbols secrecy (proper fs privs., dmesg, /proc)
Countermeasures (cont’d)

CFA motion (key technology #3; diversification)

- Relocate the injected guards & protected branches
- Make it harder for an attacker to find a bypass trampoline
kGuard Implementation

Fun times with GCC

- Implemented kGuard as a set of modifications to the pipeline of GCC ("de-facto" compiler for Linux, BSD, Android, ...)  
- Back-end plugin $\rightarrow \sim 1$KLOC in C
# kGuard Evaluation

## Effectiveness

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Description</th>
<th>Impact</th>
<th>Exploit</th>
<th>x86</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2009-1897</td>
<td>NULL function pointer</td>
<td>2.6.30–2.6.30.1</td>
<td>✓</td>
<td>✓</td>
<td>N/A</td>
</tr>
<tr>
<td>CVE-2009-2692</td>
<td>NULL function pointer</td>
<td>2.6.0–2.6.30.4</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2009-2908</td>
<td>NULL data pointer</td>
<td>≤ 2.6.31</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2009-3547</td>
<td>data pointer corruption</td>
<td>≤ 2.6.32-rc6</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2010-2959</td>
<td>function pointer overwrite</td>
<td>2.6.{27.x, 32.x, 35.x}</td>
<td>✓</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>CVE-2010-4258</td>
<td>function pointer overwrite</td>
<td>≤ 2.6.36.2</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>EDB-15916</td>
<td>NULL function pointer overwrite</td>
<td>≤ 2.6.34</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2009-3234</td>
<td>ret2usr via kernel stack buffer overflow</td>
<td>2.6.31</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

✓: detected and prevented successfully, N/A: exploit unavailable
## kGuard Evaluation

### Effectiveness

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Description</th>
<th>Impact</th>
<th>Exploit</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2009-1897</td>
<td>NULL function pointer</td>
<td>2.6.30–2.6.30.1</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2009-2692</td>
<td>NULL function pointer</td>
<td>2.6.0–2.6.30.4</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2009-2908</td>
<td>NULL data pointer</td>
<td>≤ 2.6.31</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2009-3547</td>
<td>data pointer corruption</td>
<td>≤ 2.6.32-rc6</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2010-2959</td>
<td>function pointer overwrite</td>
<td>2.6.{27.x, 32.x, 35.x}</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2010-4258</td>
<td>function pointer overwrite</td>
<td>≤ 2.6.36.2</td>
<td>✓</td>
</tr>
<tr>
<td>EDB-15916</td>
<td>NULL function pointer overwrite</td>
<td>≤ 2.6.34</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2009-3234</td>
<td>ret2usr via kernel stack buffer overflow</td>
<td>2.6.31</td>
<td>✓</td>
</tr>
</tbody>
</table>

✓: detected and prevented successfully, N/A: exploit unavailable
**kGuard Evaluation (cont’d)**

Macro benchmarks

<table>
<thead>
<tr>
<th>App. (Bench.)</th>
<th>x86</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel build (time(1))</td>
<td>1.03%</td>
<td>0.93%</td>
</tr>
<tr>
<td>MySQL (sql-bench)</td>
<td>0.92%</td>
<td>0.85%</td>
</tr>
<tr>
<td>Apache (ApacheBench)</td>
<td>≤0.01%</td>
<td>≤0.01%</td>
</tr>
</tbody>
</table>

Impact on real-life applications: \( \sim 1\% \)
Summary

**kGuard**

- Versatile & lightweight mechanism against \texttt{ret2usr} attacks
- Builds upon inline monitoring and code diversification
  - Control-flow assertions (CFAs)
  - Code inflation & CFA motion
- Cross-platform solution
  - x86, x86-64, ARM, ...
  - Linux, Android, \{Free, Open\} BSD, ...
- Non-intrusive & low overhead
  - \(\sim 1\%\) on real-life applications
ret2usr Defenses

State of the art overview

- **KERNEEXEC/UDEREF → PaX**
  - 3rd-party Linux patch(es) → x86-64/x86/AArch32 only
  - HW/SW-assisted address space separation
    - x86 → Seg. unit (reload %cs, %ss, %ds, %es)
    - x86-64 → Code instr. & temporary user space re-mapping
    - ARM (AArch32) → ARM domains

- **SMEP/SMAP, PXN/PAN →**
  - HW-assisted address space separation
    - Access violation if priv. code (ring 0) executes/accesses instructions/data from user pages (U/S = 1)
  - Vendor and model specific (Intel x86/x86-64, ARM)
ret2usr Defenses
State of the art overview
Rethinking Kernel Isolation [USENIX Sec ’14, BHEU ’14]

Kernel isolation is hard

Focus on \texttt{ret2usr} defenses $\rightarrow$ SMEP/SMAP, PXN/PAN, PaX, kGuard
Rethinking Kernel Isolation [USENIX Sec ’14, BHEU ’14]

Kernel isolation is hard

Focus on \texttt{ret2usr} defenses $\rightarrow$ SMEP/SMAP, PXN/PAN, PaX, kGuard

- Can we subvert them?
  - Force the kernel to execute/access user-controlled code/data
- Conflicting design choices or optimizations?
  - “Features” that weaken the (strong) separation of address spaces
Rethinking Kernel Isolation [USENIX Sec ’14, BHEU ’14]

Kernel isolation is hard

Focus on \texttt{ret2usr} defenses $\rightarrow$ SMEP/SMAP, PXN/PAN, PaX, kGuard

- Can we subvert them?
  - Force the kernel to execute/access user-controlled code/data
- Conflicting design choices or optimizations?
  - “Features” that weaken the (strong) separation of address spaces

Return-to-direct-mapped memory (\texttt{ret2dir})

- Attack against hardened (Linux) kernels
  - Bypasses \textbf{all} existing \texttt{ret2usr} protection schemes
  - $\forall$ \texttt{ret2usr} exploit $\rightarrow$ $\exists$ \texttt{ret2dir} exploit
The **ret2dir** Attack

Operation

- **Corrupted Code Pointer**
- **Corrupted Data Pointer**
- **physmap**
- **Controlled Data Structure**
- **Controlled Code Pointer**
- **Shellcode**
- **Kernel Space**
- **User Space**
- **Controlled Data Structure**
- **Controlled Code Pointer**
- **Shellcode**

Virtual address aliases

Direct Mapping

Physical Memory

Controlled Data Structure

Controlled Code Pointer

Shellcode
The ret2dir Attack (cont’d)

Impact

Press


Research

- ![Xen](https://goo.gl/5e73)
  - Aaron Adams, NCC Group [https://goo.gl/QKDFUp](https://goo.gl/QKDFUp)
  - “Xen SMEP (and SMAP) bypass”

- ![Android](https://goo.gl/5e73)
  - Wen Xu & Yubin Fu, Keen Team [https://goo.gl/iwp3Lk](https://goo.gl/iwp3Lk)
  - “Own Your Android! Yet Another Universal Root”
The ret2dir Attack (cont’d)

Impact

Industry

- **Qualcomm**
  - ✓ Non-executable physmap on MSM Android ⇔ http://goo.gl/NL0L3D

- **OpenBSD**
  - ✓ Non-executable pmap on x86-64, PPC ⇔ http://goo.gl/vskTwA

- ✓ Restrict (?) /proc/<pid>/pagemap ⇔ https://goo.gl/ctMy8R
Summary

- **Kernel isolation is hard**
  - × Shared kernel/process model $\rightarrow$ ret2usr
  - × physmap region(s) in kernel space $\rightarrow$ ret2dir
  - × ... $\rightarrow$ ret2...?
Summary

- Kernel isolation is hard
  - $\times$ Shared kernel/process model $\rightarrow$ ret2usr
  - $\times$ physmap region(s) in kernel space $\rightarrow$ ret2dir
  - $\times$ ... $\rightarrow$ ret2...?

- **kGuard** $\rightsquigarrow$ Versatile & lightweight mechanism against ret2usr
  - $\sim$ 1% on real-life applications
Summary

- **Kernel isolation is hard**
  - ✗ Shared kernel/process model → ret2usr
  - ✗ physmap region(s) in kernel space → ret2dir
  - ✗ ... → ret2...?

- **kGuard** → Versatile & lightweight mechanism against ret2usr protections
  - ~ 1% on real-life applications

- **ret2dir** → Deconstructs the isolation guarantees of ret2usr protections (SMEP/SMAP, PXN, PaX, kGuard)
Summary

- Kernel isolation is hard
  - × Shared kernel/process model → ret2usr
  - × physmap region(s) in kernel space → ret2dir
  - × ... → ret2...?

- kGuard ↪ Versatile & lightweight mechanism against ret2usr
  - ~ 1% on real-life applications

- ret2dir ↪ Deconstructs the isolation guarantees of ret2usr protections (SMEP/SMAP, PXN, PaX, kGuard)

Liked today’s lecture?
Register for CSCI1951-H: Software Security and Exploitation