ret2dir: Rethinking Kernel Isolation

Presenter: Jianhua Sun

Department of Computer Science
College of William and Mary
Oct 27, 2014

Slides adapted from https://www.usenix.org/sites/default/files/conference/protected-files/sec14_slides_kemerlis.pdf
The Kernel as a Target

Why care?

Kernel attacks are becoming (more) common

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 (Pwnie Awards)   14 bugs to takedown Chrome

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

Current state of affairs (all vendors)

Kernel vulnerabilities per year

Source: National Vulnerability Database (http://nvd.nist.gov)
Threat Evolution
There’s still plenty of candy left

- The kernel is highly volatile → Sub-systems change every hour
- New features & optimizations → New attack opportunities

<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: The Linux Foundation
Return-to-user (*ret2usr*) Attacks

What are they?

**Attacks against OS kernels with shared kernel/user address space**

- Overwrite kernel code (or data) pointers with *user space* addresses
  - return addr., dispatch tbl., function ptr.,
  - data ptr.
- Payload → Shellcode, ROP payload, tampered-with data structure(s)
  - Placed in user space
  - Executed (referenced) in kernel context
- De facto kernel exploitation technique
  - Facilitates privilege escalation ⇔ arbitrary code execution
  - http://www.exploit-db.com/exploits/34134/ (21/07/14)
  - http://www.exploit-db.com/exploits/131/ (05/12/03)
ret2usr Attacks (cont’d)
Why do they work?

Weak address space (kernel/user) separation

- Shared kernel/process model $\rightarrow$ Performance
  - $\checkmark$ cost(mode\_switch) $\ll$ cost(context\_switch)

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

- The attacker completely controls user space memory
  - Contents & perms.
ret2usr Defenses

State of the art overview

**KERNEXEC/UDEREF → PaX**
- $3^{rd}$-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

**kGuard → Kemerlis et al. [USENIX Sec ’12]**
- Cross-platform solution that enforces (partial) address space separation
  - x86, x86-64, ARM, ...
  - Linux, {Free, Net, Open}BSD, ...
- Builds upon inline monitoring (code intr.) & code diversification (code inflation & CFA motion)

**SMEP/SMAP, PXN → Intel, ARM**
- 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)
Defenses (cont’d)
Rethinking Kernel Isolation

What is this work about?

**Focus on ret2usr defenses** → SMEP/SMAP, PXN, PaX, kGuard
Rethinking Kernel Isolation

What is this work about?

Focus on ret2usr defenses → SMEP/SMAP, PXN, 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
What is this work about?

Focus on \texttt{ret2usr} defenses $\rightarrow$ SMEP/SMAP, PXN, 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 all existing \texttt{ret2usr} schemes
  - $\forall \texttt{ret2usr}$ exploit $\implies \exists \texttt{ret2dir}$ exploit
Kernel Space Layout

Linux x86-64

---

**physmap**
(direct mapping of all physical memory)
physmap
Functionality

**Fundamental** building block of dynamic kernel memory
(kmalloc, SLAB/SLUB)

1. (De)allocate kernel memory **without** altering page tables
   - Minimum latency in fast-path ops. (**e.g.**, kmalloc in ISR)
   - Less TLB pressure → No TLB shootdown(s) needed
2. Virtually contiguous memory → Physically contiguous (**guaranteed**)
   - Directly assign kmalloc-ed memory to devices for DMA
   - Increased cache performance
3. Page frame accounting made easy
   - `virt(pfn)`  `PHYS_OFFSET + (pfn << PAGE_SHIFT)`
   - `pfn(vaddr)`  `(vaddr - PHYS_OFFSET) >> PAGE_SHIFT`
### physmap
Location, size, and access rights

<table>
<thead>
<tr>
<th>Architecture</th>
<th>PHYS.OFFSET</th>
<th>Size</th>
<th>Prot.</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(3G/1G)</td>
<td>0xC00000000</td>
<td>891MB</td>
<td>RW</td>
</tr>
<tr>
<td>(2G/2G)</td>
<td>0x80000000</td>
<td>1915MB</td>
<td>RW</td>
</tr>
<tr>
<td>(1G/3G)</td>
<td>0x40000000</td>
<td>2939MB</td>
<td>RW</td>
</tr>
<tr>
<td>AArch32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(3G/1G)</td>
<td>0xC00000000</td>
<td>760MB</td>
<td>RW(X)</td>
</tr>
<tr>
<td>(2G/2G)</td>
<td>0x80000000</td>
<td>1784MB</td>
<td>RW(X)</td>
</tr>
<tr>
<td>(1G/3G)</td>
<td>0x40000000</td>
<td>2808MB</td>
<td>RW(X)</td>
</tr>
<tr>
<td>x86-64</td>
<td>0xFFFF88000000000000</td>
<td>64TB</td>
<td>RW(X)</td>
</tr>
<tr>
<td>AArch64</td>
<td>0xFFFFFFF000000000000</td>
<td>256GB</td>
<td>RW(X)</td>
</tr>
</tbody>
</table>
The *ret2dir* Attack

*physmap* is considered harmful

*physmap* $\leadsto$ **Address aliasing**

*Given the existence of* *physmap*, whenever the kernel (*buddy allocator*) maps a page frame to user space, it effectively creates an *alias* (*synonym*) of user content in kernel space!
The ret2dir Attack (cont’d)

Operation

- Corrupted Code Pointer
- Corrupted Data Pointer
- Controlled Data Structure
- Controlled Code Pointer
- Shellcode
- Kernel Space
- User Space
- Controlled Data Structure
- Controlled Code Pointer
- Shellcode
- Virtual Memory

Direct Mapping

- Controlled Data Structure
- Controlled Code Pointer
- Shellcode
- Physical Memory
Locating Synonyms
Leaking PFNs via /proc (1/2)

\( C_1 \): Given a user space virtual address (\texttt{uaddr}) \rightarrow \text{Synonym in kernel space (\texttt{kaddr})}

- Usual suspect: /proc (procfs)
  - /proc/<pid>/pagemap \rightarrow \text{Page table examination (from user space) for debugging purposes (since v2.6.25)}
    - 64-bit value per page \rightarrow \text{Indexed by virtual page number}
      - [0:54] \rightarrow \text{Page frame number (PFN)}
      - [63] \rightarrow \text{Page present}

\textbf{PFN (\texttt{uaddr})}

```c
seek( (uaddr >> PAGE_SHIFT) * sizeof(uint64_t));
read( &v, sizeof(uint64_t));
if (v & (1UL << 63))
    PFN = v & ((1UL << 55) - 1);
```
Locating Synonyms (cont’d)
Leaking PFNs via `/proc` (2/2)

\[ F_1: kaddr = \text{PHYS\_OFFSET} + \text{PAGE\_SIZE} \times (\text{PFN}(uaddr) - \text{PFN\_MIN}) \]

- **PHYS\_OFFSET** → Starting address of `physmap` in kernel space
- **PFN\_MIN** → 1\(^{st}\) PFN (e.g., in ARM Versatile RAM starts at 0x60000000; PFN\_MIN = 0x60000)

<table>
<thead>
<tr>
<th>Architecture</th>
<th>PHYS_OFFSET</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86</td>
<td></td>
</tr>
<tr>
<td>(3G/1G)</td>
<td>0xC0000000</td>
</tr>
<tr>
<td>(2G/2G)</td>
<td>0x80000000</td>
</tr>
<tr>
<td>(1G/3G)</td>
<td>0x40000000</td>
</tr>
<tr>
<td>AArch32</td>
<td></td>
</tr>
<tr>
<td>(3G/1G)</td>
<td>0xC0000000</td>
</tr>
<tr>
<td>(2G/2G)</td>
<td>0x80000000</td>
</tr>
<tr>
<td>(1G/3G)</td>
<td>0x40000000</td>
</tr>
<tr>
<td>x86-64</td>
<td>0xFFFFF8800000000000</td>
</tr>
<tr>
<td>AArch64</td>
<td>0xFFFFFFFFFC000000000</td>
</tr>
</tbody>
</table>
Locating Synonyms (cont’d)

ret2dir without access to /proc/<pid>/pagemap

Q: What ifPFN information is not available?
Locating Synonyms (cont’d)
ret2dir without access to /proc/<pid>/pagemap

Q: What if PFN information is not available?

**physmap spraying** → Very similar to how heap spraying works

1. Pollute **physmap** with **aligned** copies of the exploit payload
   - Maximize the exploit foothold on **physmap**
2. Pick an arbitrary, page-aligned **physmap** address and use it as the synonym of the exploit payload
The attacking process copies the exploit payload into $N$ physmap-resident pages.

The probability $P$ that an arbitrarily chosen, page-aligned physmap address will contain the exploit payload is: $P = \frac{N}{(PFN_{\text{MAX}} - PFN_{\text{MIN}})}$.
The attacking process copies the exploit payload into $N$ physmap-resident pages.

The probability $P$ that an arbitrarily chosen, page-aligned physmap address will contain the exploit payload is:

$$P = \frac{N}{PFN_{\text{MAX}} - PFN_{\text{MIN}}}$$

$max(P)$

1. $max(N)$

2. $min(PFN_{\text{MAX}} - PFN_{\text{MIN}})$
physmap Spraying
max(N)

1. Allocate a (big) chunk of RW memory in user space → $M$
   ▶ mmap/mmap2, shmat, ...

2. $\forall$ page $P \in M$ → Copy the exploit payload in $P$ and trigger a \texttt{write} fault (or MAP_POPULATE)

3. “Emulate” \texttt{mlock} → Prevent swapping
   ▶ Start a set of background threads that repeatedly mark payload pages as \texttt{dirty} (e.g., by writing a single byte)

4. Check RSS (foothold in physmap) → getrusage

5. goto 1, unless $RSS < RSS_{prev}$

   • If sizeof(uspace) $\ll$ sizeof(RAM) → Spawn additional process(es)
Reduce the set of target pages in `physmap` → **physmap signatures**

- **x86**
  - Page frame 0 is used by BIOS → HW config. discovered during POST
  - [0xA0000:0xFFFFF] → Memory-mapped RAM of video cards
- **x86-64**
  - 0x1000000 → Kernel .text, .rodata, data, .bss
- **AArch32**
  - ...
- **AArch64**
  - ...
Evaluation

Spraying performance

- 2x 2.66GHz quad core Xeon X5500, 16GB RAM, 64-bit Debian Linux v7
- 5 repetitions of the same experiment, 95% confidence intervals (error bars)
Defending against \textit{ret2dir} Attacks

Design

\textbf{eXclusive Page Frame Ownerwhip (XPFO)}

- Thin mgmt. layer over the buddy allocator $\rightarrow$ Exclusive ownership (of page frames) by \textbf{either} the kernel or userland
  - Unless explicitly requested by a kernel component (e.g., to implement zero-copy buffers)

1. Page frame(s) allotted to userland $\rightarrow$ Synonym page(s) unmapped from \texttt{physmap}
2. Page frame(s) reclaimed from userland $\rightarrow$ Synonym page(s) put back to \texttt{physmap}
  - Reclaimed page frames are always \textbf{wiped out} before remapping

- Performance-critical kernel allocators are \textbf{not} affected $\rightarrow$ Low extra overhead whenever page frames are allotted to (or reclaimed from) user processes
  - Aligns well with demand paging & COW
Defending against `ret2dir` Attacks
Implementation (1/2)

XPFO \(\sim\) Linux kernel v3.13 (\(\sim\)500LOC)

- `struct page` extended with XPFO fields +3MB per 1GB of RAM
  - `xpfo_kmcnt` (ref. counter), `xpfo_lock` (spinlock), `xpfo_flags`
- Careful handling of page frame allocation/reclamation cases
  - Demand paging frames (anonymous & shared memory mappings)
    - [stack], `brk`, `mmap/mmap2`, `mremap`, `shmat`
  - COW frames
    - `fork`, `clone`
  - Explicitly & implicitly reclaimed frames
    - `exit`, `munmap`, `shmdt`
  - Swapping (swapped out and swapped in pages)
  - NUMA frame migrations
    - `migrate_pages`, `move_pages`
  - Huge pages & transparent huge pages
### Evaluation

**ret2dir effectiveness**

<table>
<thead>
<tr>
<th>EDB-ID</th>
<th>Arch.</th>
<th>Kernel</th>
<th>Payload</th>
<th>Protection</th>
<th>Bypassed</th>
</tr>
</thead>
<tbody>
<tr>
<td>26131</td>
<td>x86/x64</td>
<td>3.5/3.8</td>
<td>ROP/SHELLCODE</td>
<td>KERNEXEC</td>
<td>UDEREF</td>
</tr>
<tr>
<td>24746</td>
<td>x86-64</td>
<td>3.5</td>
<td>SHELLCODE</td>
<td>KERNEXEC</td>
<td></td>
</tr>
<tr>
<td>15944</td>
<td>x86</td>
<td>2.6.33.6</td>
<td>STRUCT+ROP</td>
<td>KERNEXEC</td>
<td>UDEREF</td>
</tr>
<tr>
<td>15704</td>
<td>x86</td>
<td>2.6.35.8</td>
<td>STRUCT+ROP</td>
<td>KERNEXEC</td>
<td>UDEREF</td>
</tr>
<tr>
<td>15285</td>
<td>x86-64</td>
<td>2.6.33.6</td>
<td>ROP/SHELLCODE</td>
<td>KERNEXEC</td>
<td>UDEREF</td>
</tr>
<tr>
<td>15150</td>
<td>x86</td>
<td>2.6.35.8</td>
<td>STRUCT</td>
<td></td>
<td>UDEREF</td>
</tr>
<tr>
<td>15023</td>
<td>x86-64</td>
<td>2.6.33.6</td>
<td>STRUCT+ROP</td>
<td>KERNEXEC</td>
<td>UDEREF</td>
</tr>
<tr>
<td>14814</td>
<td>x86</td>
<td>2.6.33.6</td>
<td>STRUCT+ROP</td>
<td>KERNEXEC</td>
<td>UDEREF</td>
</tr>
<tr>
<td>Custom</td>
<td>x86</td>
<td>3.12</td>
<td>STRUCT+ROP</td>
<td>KERNEXEC</td>
<td>UDEREF</td>
</tr>
<tr>
<td>Custom</td>
<td>x86-64</td>
<td>3.12</td>
<td>STRUCT+ROP</td>
<td>KERNEXEC</td>
<td>UDEREF</td>
</tr>
<tr>
<td>Custom</td>
<td>AArch32</td>
<td>3.8.7</td>
<td>STRUCT+SHELLCODE</td>
<td>KERNEXEC</td>
<td>UDEREF</td>
</tr>
<tr>
<td>Custom</td>
<td>AArch64</td>
<td>3.12</td>
<td>STRUCT+SHELLCODE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Evaluation

**ret2dir effectiveness**

<table>
<thead>
<tr>
<th>EDB-ID</th>
<th>Arch.</th>
<th>Kernel</th>
<th>Payload</th>
<th>Protection</th>
<th>Bypassed</th>
</tr>
</thead>
<tbody>
<tr>
<td>26131</td>
<td>x86/x86-64</td>
<td>3.5/3.8</td>
<td>ROP/SHELLCODE</td>
<td>KERNEXEC</td>
<td>UDEREF</td>
</tr>
<tr>
<td>24746</td>
<td>x86-64</td>
<td>3.5</td>
<td>SHELLCODE</td>
<td>KERNEXEC</td>
<td></td>
</tr>
<tr>
<td>15944</td>
<td>x86</td>
<td>2.6.33.6</td>
<td>STRUCT+ROP</td>
<td>KERNEXEC</td>
<td>UDEREF</td>
</tr>
<tr>
<td>15704</td>
<td>x86</td>
<td>2.6.35.8</td>
<td>STRUCT+ROP</td>
<td>KERNEXEC</td>
<td>UDEREF</td>
</tr>
<tr>
<td>15285</td>
<td>x86-64</td>
<td>2.6.33.6</td>
<td>ROP/SHELLCODE</td>
<td>KERNEXEC</td>
<td>UDEREF</td>
</tr>
<tr>
<td>15150</td>
<td>x86</td>
<td>2.6.35.8</td>
<td>STRUCT</td>
<td></td>
<td>UDEREF</td>
</tr>
<tr>
<td>15023</td>
<td>x86-64</td>
<td>2.6.33.6</td>
<td>STRUCT+ROP</td>
<td>KERNEXEC</td>
<td>UDEREF</td>
</tr>
<tr>
<td>14814</td>
<td>x86</td>
<td>2.6.33.6</td>
<td>STRUCT+ROP</td>
<td>KERNEXEC</td>
<td>UDEREF</td>
</tr>
<tr>
<td>Custom</td>
<td>x86</td>
<td>3.12</td>
<td>STRUCT+ROP</td>
<td>KERNEXEC</td>
<td>UDEREF</td>
</tr>
<tr>
<td>Custom</td>
<td>x86-64</td>
<td>3.12</td>
<td>STRUCT+ROP</td>
<td>KERNEXEC</td>
<td>UDEREF</td>
</tr>
<tr>
<td>Custom</td>
<td>AArch32</td>
<td>3.8.7</td>
<td>STRUCT+SHELLCODE</td>
<td>KERNEXEC</td>
<td>UDEREF</td>
</tr>
<tr>
<td>Custom</td>
<td>AArch64</td>
<td>3.12</td>
<td>STRUCT+SHELLCODE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Evaluation

### XPFO performance

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Metric</th>
<th>Original</th>
<th>XPFO (%)Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>Req/s</td>
<td>17636.30</td>
<td>17456.47 (%1.02)</td>
</tr>
<tr>
<td>NGINX</td>
<td>Req/s</td>
<td>16626.05</td>
<td>16186.91 (%2.64)</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>Trans/s</td>
<td>135.01</td>
<td>134.62 (%0.29)</td>
</tr>
<tr>
<td>Kbuild</td>
<td>sec</td>
<td>67.98</td>
<td>69.66 (%2.47)</td>
</tr>
<tr>
<td>Kextract</td>
<td>sec</td>
<td>12.94</td>
<td>13.10 (%1.24)</td>
</tr>
<tr>
<td>GnuPG</td>
<td>sec</td>
<td>13.61</td>
<td>13.72 (%0.80)</td>
</tr>
<tr>
<td>OpenSSL</td>
<td>Sign/s</td>
<td>504.50</td>
<td>503.57 (%0.18)</td>
</tr>
<tr>
<td>PyBench</td>
<td>ms</td>
<td>3017.00</td>
<td>3025.00 (%0.26)</td>
</tr>
<tr>
<td>PHPBench</td>
<td>Score</td>
<td>71111.00</td>
<td>70979.00 (%0.18)</td>
</tr>
<tr>
<td>IOZone</td>
<td>MB/s</td>
<td>70.12</td>
<td>69.43 (%0.98)</td>
</tr>
<tr>
<td>tiobench</td>
<td>MB/s</td>
<td>0.82</td>
<td>0.81 (%1.22)</td>
</tr>
<tr>
<td>dbench</td>
<td>MB/s</td>
<td>20.00</td>
<td>19.76 (%1.20)</td>
</tr>
<tr>
<td>PostMark</td>
<td>Trans/s</td>
<td>411.00</td>
<td>399.00 (%2.91)</td>
</tr>
</tbody>
</table>
Summary

physmap is considered harmful

- physmap region(s) in kernel space is a bad idea
  - ret2dir Deconstructs the isolation guarantees of ret2usr protections (SMEP/SMAP, PXN, PaX, kGuard)
  - XPFO Low overhead defense against ret2dir attacks
Q & A