ret2dir: Rethinking Kernel Isolation

Vasileios P. Kemerlis
Michalis Polychronakis    Angelos D. Keromytis

Network Security Lab
Department of Computer Science
Columbia University
New York, NY, USA

USENIX Security Symposium
Return-to-user (\textit{ret2usr}) Attacks

What are they?

Attacks against OS kernels with \textcolor{red}{\textbf{shared}} kernel/user address space

- Overwrite kernel code (or data) pointers with \textbf{user space} addresses
  - return addr., dispatch tbl., function ptr.,
  - data ptr.
- Payload $\rightarrow$ Shellcode, ROP payload, tampered-with data structure(s)
  - Placed in user space
  - Executed (referenced) in \textcolor{red}{\textbf{kernel}} context
- De facto kernel exploitation technique
  - Facilitates privilege escalation $\leadsto$ arbitrary code execution
    - \url{http://www.exploit-db.com/exploits/34134/} (21/07/14)
    - \url{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**
  - ✓ cost(mode_switch) ≪ cost(context_switch)

- The kernel is protected from userland → Hardware-assisted isolation
  - ✗ The opposite is **not** true
  - ✗ 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**
  ▶ 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

✓ **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
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 $\rightarrow$ \texttt{SMEP}/\texttt{SMAP}, \texttt{PXN}, \texttt{PaX}, \texttt{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 $\rightarrow$ $\exists$ \texttt{ret2dir} exploit
Kernel Space Layout

Linux x86-64

---

vpk@cs.columbia.edu  (Columbia University)  ret2dir  USENIX Sec ’14  7 / 40
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
   - \( \text{virt} (\text{pf}n) \mapsto \text{PHYS\_OFFSET} + (\text{pf}n \ll \text{PAGE\_SHIFT}) \)
   - \( \text{pf}n (\text{vaddr}) \mapsto (\text{vaddr} - \text{PHYS\_OFFSET}) \gg \text{PAGE\_SHIFT} \)
The \texttt{ret2dir} Attack

\texttt{physmap} is considered harmful

\texttt{physmap} $\leadsto$ \textbf{Address aliasing}

\textit{Given the existence of \texttt{physmap}, whenever the kernel (buddy allocator) maps a page frame to user space, it effectively creates an \textit{alias} (\textit{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

Direct Mapping

Controlled Data Structure
Controlled Code Pointer
Shellcode

Virtual Memory

Physical Memory
Locating Synonyms
Leaking PFNs via `/proc` (1/2)

**C₁**: 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)

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

- \textbf{PHYS\_OFFSET} → Starting address of physmap in kernel space
- \textbf{PFN\_MIN} → 1\textsuperscript{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>0xFFFFF880000000000</td>
</tr>
<tr>
<td>AArch64</td>
<td>0xFFFFF7FC00000000</td>
</tr>
</tbody>
</table>

vpk@cs.columbia.edu (Columbia University)
Locating Synonyms (cont’d)

Q: What if PFN 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
# 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></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></td>
<td>Custom</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>kGuard</td>
</tr>
</tbody>
</table>

vpk@cs.columbia.edu  (Columbia University)
Defending against \texttt{ret2dir} Attacks

Design

\textbf{eXclusive Page Frame Ownerwhip (XPFO)}

\begin{itemize}
  \item Thin mgmt. layer over the buddy allocator \implies Exclusive ownership (of page frames) by \textit{either} the kernel or userland
    \begin{itemize}
      \item Unless explicitly requested by a kernel component (\eg, to implement zero-copy buffers)
    \end{itemize}
  \end{itemize}

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

\begin{itemize}
  \item Performance-critical kernel allocators are \textbf{not} affected \implies Low extra overhead whenever page frames are allotted to (or reclaimed from) user processes
    \begin{itemize}
      \item Aligns well with demand paging & COW
    \end{itemize}
\end{itemize}
## 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

Code (ret2dir exploits & XPFO patch)

http://www.cs.columbia.edu/~vpk/research/ret2dir/
Bonus Slides
The Kernel as a Target

Why care?

Kernel attacks are becoming (more) common

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

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

3. Exploiting privileged userland processes has become harder $\rightarrow$
   Canaries+ASLR+W^X+Fortify+RELRO+BIND_NOW+BPF_SECCOMP+...
   - Sergey Glazunov (Pwnie Awards) $\Rightarrow$ 14 bugs to takedown Chrome

“A Tale of Two Pwnies” (http://blog.chromium.org)
Attacking the “Core”

Threats classification

1. Privilege escalation
   - Arbitrary code execution $\leadsto$ code-injection, ROP, \texttt{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

3. ...
Kernel Vulnerabilities
Current state of affairs (all vendors)

Kernel vulnerabilities per year

# of vulnerabilities

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

Current state of affairs (Linux only)

Linux kernel vulnerabilities per year

Source: CVE Details (http://www.cvedetails.com)
vpk@cs.columbia.edu (Columbia University)
Threat Evolution

What's next?

✓ SMEP/SMAP, PXN, KERNEXEC/UDEREF, kGuard ↦ ret2usr
✓ KASLR, W^X, stack canaries, SLAB red zones, const dispatch tbl., .rodata sections, ... ↦ Traditional (kernel) exploitation

What will next generation kernel exploits do?

► ROP-based code execution?
► Data-only attacks?
► Subvert hardening mechanisms by chaining together component-specific vulnerabilities?
Threat Evolution (cont’d)
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
Kernel Space Layout

Linux x86/x86-64

- Physical Memory
  - vmap arena
  - pkmap space
  - fixmap/vsyscall area
  - kernel image
  - modules
  - vmemmap space
  - vmalloc arena
  - unused

- Upper Wordmark
  - Complex Memory Layout

- Lower Wordmark
  - Fix Page Table
  - Kernel Space Layout
  - Complex Memory Layout
Kernel Space Layout (cont’d)

Solaris sun4u

0x0FFFFF00,0FFFFF00
0x0FFFFF08,0FFFFF00
0x0FFFFF08,0FFFFF00
0x0FFFFF80,0FFFFF00
0x0FFFFF80,0FFFFF00
0x00000000,00000000
0x00000000,00000000
0x00000000,00000000
0x00000000,00000000
0x00000000,00000000
0x00000000,00000000
0x00000000,00000000
0x00000000,00000000

- Open Boot Prom
- Page Tables

- Physical Page Mapping
- segkpm

- 64-Bit Kernel Heap
- segkmem

- File System Cache
- segmap

- Pageable Kernel Mem.
- segkp

- Open Boot Prom

- Kernel Debugger

- 32-Bit Kernel Heap
- segkmem32

- Panic Message Buffer

- Kernel TSB

- sun4u HAT Structures
- Small TSB & Map Blks

- Kernel Data Segment

- Kernel Text Segment

- Trap Table

- Invalid

Source: Solaris Internals (2nd ed.)

vpk@cs.columbia.edu (Columbia University)
Kernel Space Layout (cont’d)

Solaris x86-64

Source: Solaris Internals (2nd ed.)
vpk@cs.columbia.edu (Columbia University)
## 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>0xFFFF880000000000</td>
<td>64TB</td>
<td>RW(X)</td>
</tr>
<tr>
<td>AArch64</td>
<td>0xFFFFFFC0000000000</td>
<td>256GB</td>
<td>RW(X)</td>
</tr>
</tbody>
</table>
The **ret2dir** Attack

The devil is (always) in the detail

**Challenges**

1. Pinpoint the *exact* location of a synonym of user-controlled data (payload) within the **physmap area**
2. When $\text{sizeof(physmap)} < \text{sizeof(RAM)} \rightarrow$ Force a synonym of payload to **emerge inside the physmap area**
3. When $\text{sizeof(payload)} > \text{PAGE\_SIZE} \rightarrow$ Force synonym pages to be **contiguous in physmap**
Ensuring the Presence of Synonyms

What if `sizeof(physmap) < sizeof(RAM)`?

\[ C_2: \text{Force a synonym of payload to emerge inside } \text{physmap} \]

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

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86</td>
<td></td>
</tr>
<tr>
<td>(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</td>
<td></td>
</tr>
<tr>
<td>(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 \leq 16MB
- ZONE_DMA < ZONE_NORMAL \leq \min(\text{sizeof(physmap)}, \text{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: ZONE_DMA < · ZONE_NORMAL < · ZONE_HIGHMEM
- User space gets page frames from 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 sizeof(physmap) < sizeof(RAM)?

\( C_2 \): Force a synonym of payload to emerge inside physmap

1. Allocate a (big) chunk of \( \text{RW} \) memory in user space \( \rightarrow M \)
   - \( \text{mmap/mmap2, shmat, ...} \)
2. \( \forall \text{ page } P \in M \rightarrow \text{Trigger a write fault (or MAP\_POPULATE)} \)
3. \( \text{If } \exists P \in M, \text{PFN}(P) \leq \text{PFN\_MAX} \)
   - \( \text{mlock(P)} \)
   - \( \text{Compute kaddr using } F_1(P) \)
4. Else, goto 1

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

What if sizeof(payload) > PAGE_SIZE?

C₃: Force synonym pages to be contiguous in physmap

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

2. ∀ page P ∈ M → Trigger a write fault (or MAP_POPULATE)

3. If ∃Pᵢ, Pⱼ ∈ M, PFN(Pⱼ) = PFN(Pᵢ) + 1
   ▶ mlock(Pᵢ, Pⱼ)
   ▶ Split the payload in Pᵢ & Pⱼ (synonyms of Pᵢ, Pⱼ are contiguous)
   ▶ Compute kaddr using F₁(min(Pᵢ, Pⱼ))

4. Else, goto 1

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

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:

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

physmap spraying

- 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: $P = N / (PFN_{MAX} - PFN_{MIN})$

$max(P)$

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

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

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

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

4. Check \text{RSS} (foothold in physmap) \rightarrow \text{getrusage}

5. \text{goto 1, unless} \( \text{RSS} < \text{RSS}_{\text{prev}} \)

- \text{If} sizeof(uspace) \ll sizeof(RAM) \rightarrow \text{Spawn additional process(es)}
physmap Spraying (cont’d)

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**
  - ...

vpk@cs.columbia.edu (Columbia University)
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)
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 ret2dir Attacks

Implementation (1/2)

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

- `struct page` extended with XPFO fields $\leadsto$ +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
Defending against ret2dir Attacks (cont’d)
Implementation (2/2)

Optimizations

1. No full TLB flush(es) $\rightsquigarrow$ Selective TLB entry invalidation(s) only (e.g., using INVLP in x86/x86-64)
2. TLB shootdown avoidance [xpfo_flags.S] $\rightsquigarrow$ Cascade TLB updates only when absolutely necessary
3. No page frame re-sanitization [xpfo_flags.Z] $\rightsquigarrow$ Avoid zeroing page frames twice (e.g., when a page frame reclaimed by a user process is subsequently allocated to a kernel path that requires a clean page)