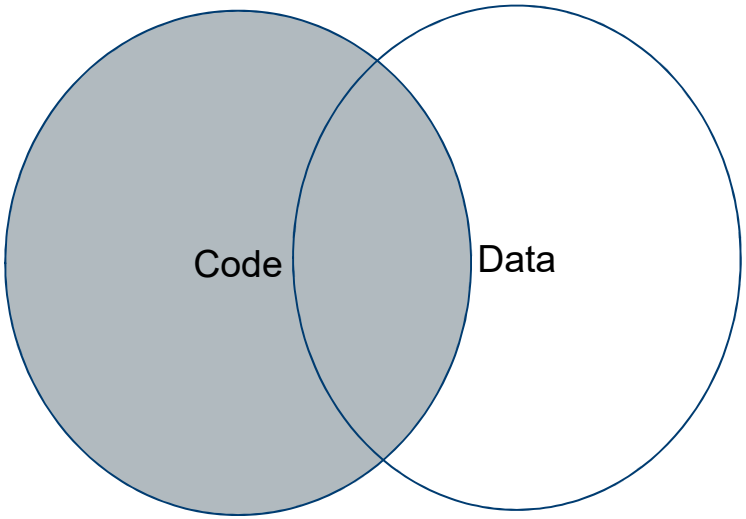
Touch but don’t look - Running the Kernel in Execute-only memory

Rick Edgecombe

Execute-only memory

- XO can make it harder to find information needed for exploits
- We can enable XO for VMs on many existing CPUs
- Path to make this safer to turn on



Why use execute-only memory

Exploits that hijack control flow need to know where things are

When the executable code (text) is secret

- Closed source
- Un-published cloud kernels (config, patches, compilers)
- Randomized
  - Fine-grained KASLR - Kristen Accardi [1]

[1] https://github.com/kaccardi/linux/wiki/Finer-grained-kernel-randomization

Control flow attacks

Control flow attacks need to know where to re-direct control flow to

Simple example: CVE-2016-0728

- Use after free for heap allocation that contains function pointers

Attacker needs to know where to redirect control flow

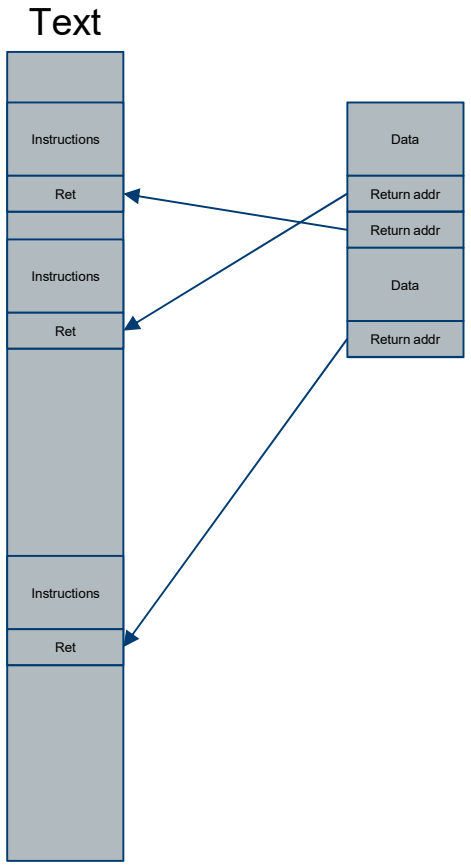
```
/*
* kernel managed key type definition
*/
struct key_type {
        ...

        /* vet a description */
        int (*vet_description)(const char *description);
        int (*preparse)(struct key_preparsed_payload *prep);

        ...
} __randomize_layout;
```

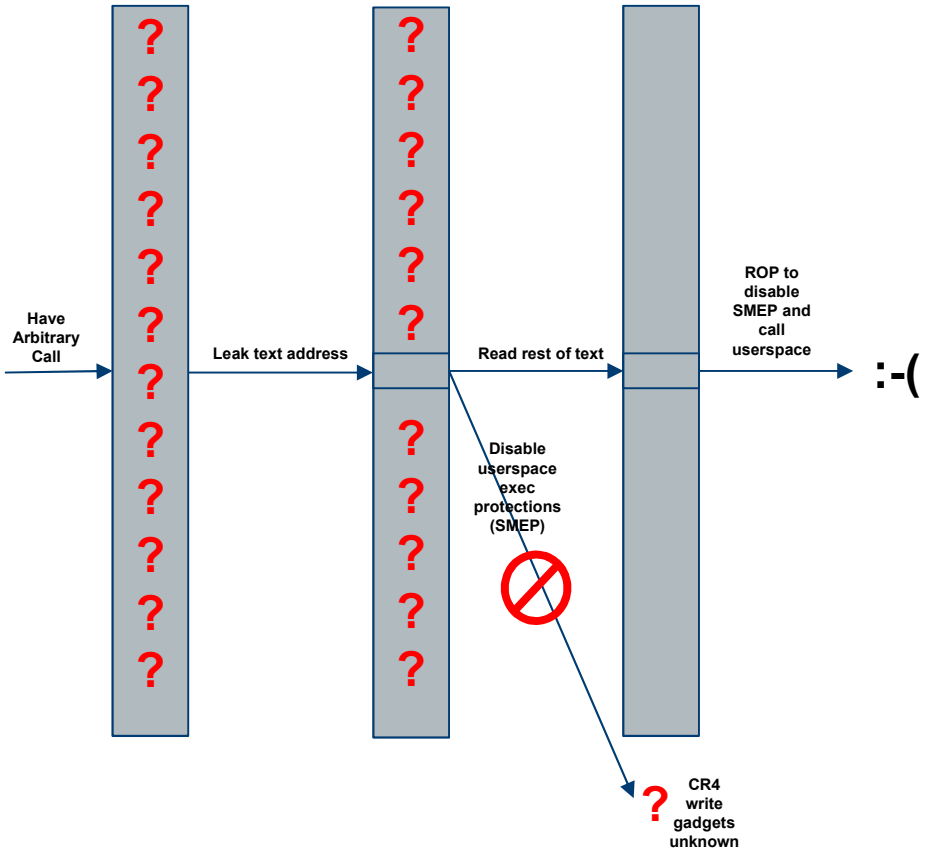
ROP

- ROP attacks need to know even more text
- Switch to fake stack with return addresses to little snippets of useful gadgets followed by a ret
- Run attacker “code” as stack unwinds despite protections on executing data



JIT-ROP

In the presence of unknown text, how can attacker find gadgets?



Discovering text - XO is not a lock box

Ways to discover text:

Leak pointer to known text
Cache side channels
Using read exploit to read text

- **This is the method we are blocking with this work**

Mitigations: Cost vs Benefit

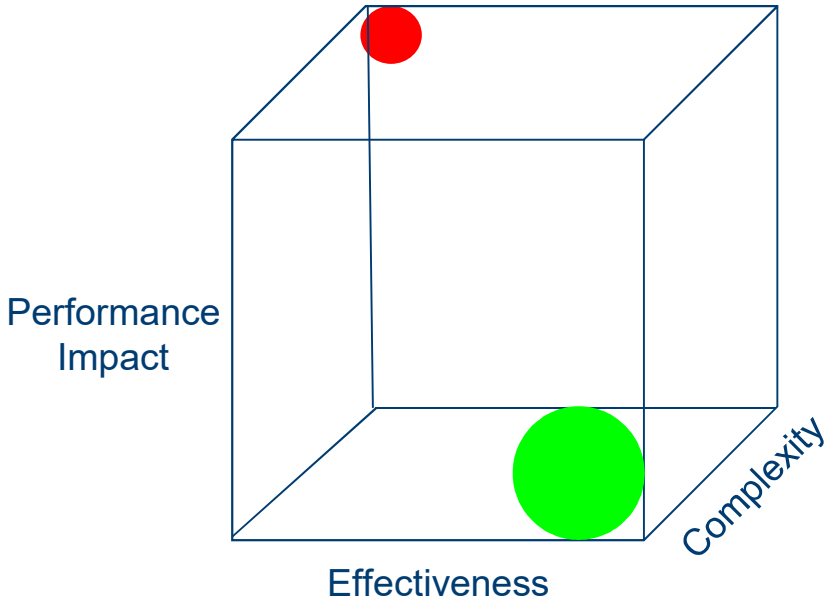
Security is increasing concern, what to do?

Mitigation tradeoffs

- Effectiveness
- Performance
- Complexity

Execute-only memory cannot stop all attacks, but...

- Negligible performance cost



Part 2: Making It Work

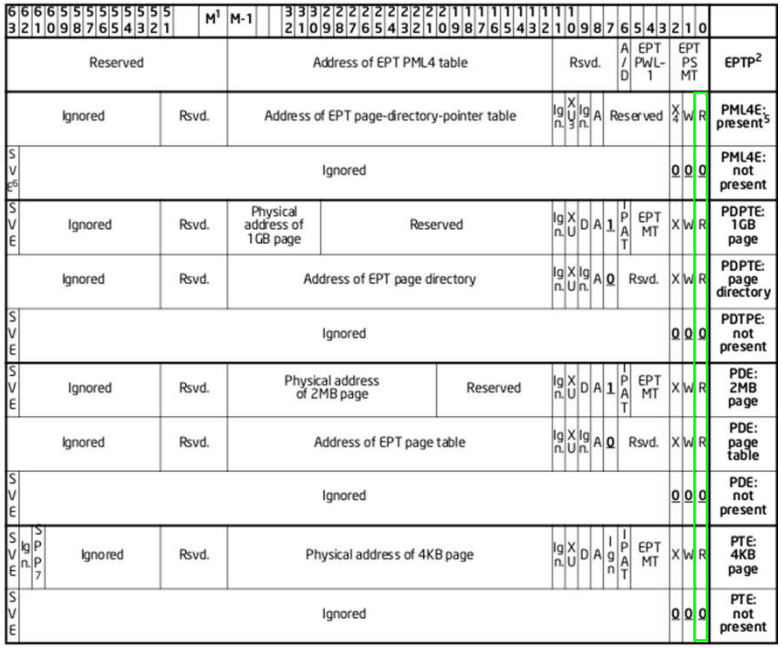
XO memory CPU support

XO Memory support: x86 pkeys, arm64, misc

- Supported in EPT on some CPUs going back generations

EPT page tables can associate permissions with *physical memory*

We can do a trick to add an XO bit to guest *virtual* page tables



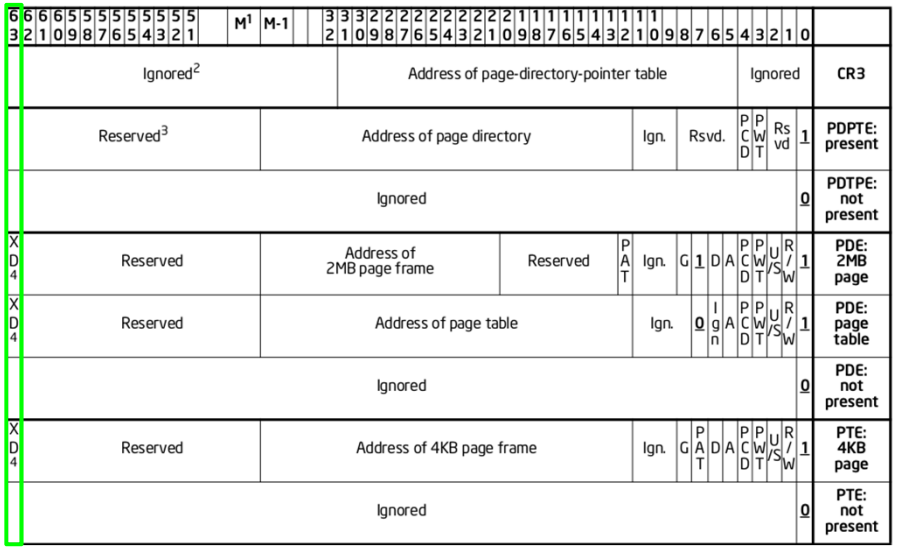
63–62	61–59	58–57	56–54	53–52	51	M[1]	M-1	32–10	9–8	7	6	5–4	3	2–1	0	
Reserved								Address of EPT PML4 table	Rsvd.		A/D	EPT PWL-1		EPT PS MT		EPTP[2]
Ignored				Rsvd.				Address of EPT page-directory-pointer table	Ign.	XU[3]	Ign.	A	Reserved		X[4] W R	PML4E: present[5]
SVE[6]	Ignored														0 0 0	PML4E: not present
SVE	Ignored			Rsvd.		Physical address of 1GB page	Reserved		Ign.	XU	D	A	1	IPAT	EPT MT	X W R	PDPTE: 1GB page
	Ignored			Rsvd.				Address of EPT page directory	Ign.	XU	Ign.	A	0	Rsvd.		X W R	PDPTE: page directory
SVE	Ignored														0 0 0	PDPTE: not present
SVE	Ignored			Rsvd.		Physical address of 2MB page		Reserved	Ign.	XU	D	A	1	IPAT	EPT MT	X W R	PDE: 2MB page
	Ignored			Rsvd.				Address of EPT page table	Ign.	XU	Ign.	A	0	Rsvd.		X W R	PDE: page table
SVE	Ignored														0 0 0	PDE: not present
SVE	Ign.	SPP[7]	Ignored		Rsvd.			Physical address of 4KB page	Ign.	XU	D	A	Ign	IPAT	EPT MT	X W R	PTE: 4KB page
SVE	Ignored														0 0 0	PTE: not present

Figure 28-1. Formats of EPTP and EPT Paging-Structure Entries

NX Bit

- In the past all memory was executable
- Attackers could jump to any data passed into the kernel
- “NX bit” introduced in amd64

- A bit combination to mark pages as execute-only fits best with existing memory protections



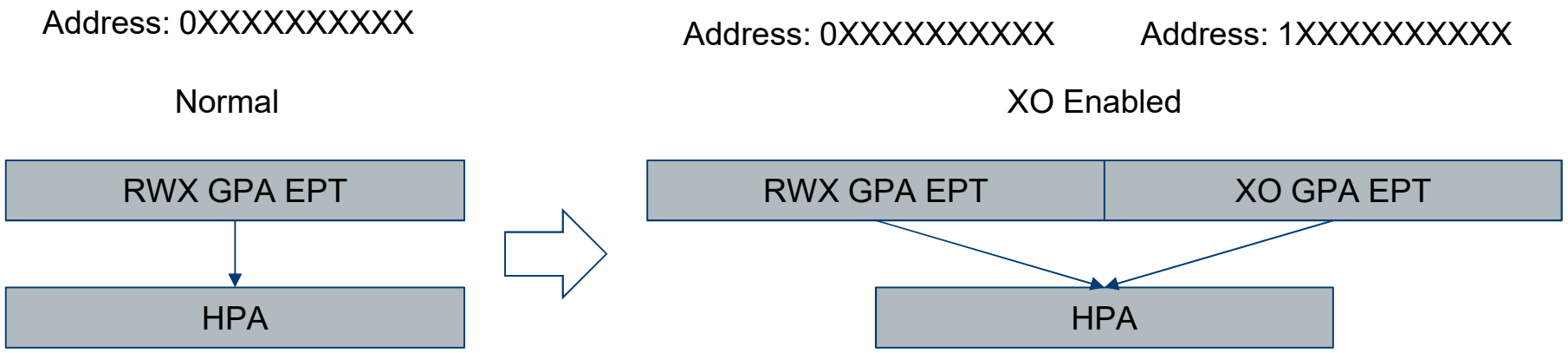
63	62–M	M-1	32–12	11–9	8	7	6	5	4	3	2	1	0	
	Ignored[2]		Address of page-directory-pointer table						Ignored					CR3
	Reserved[3]		Address of page directory			Ign.		Rsvd.		PCD	PWT	Rsvd	1	PDPTE: present
	Ignored												0	PDPTE: not present
XD	Reserved		Address of 2MB page frame		Reserved	PAT	Ign.	G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 2MB page
XD	Reserved		Address of page table				Ign.		0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table
	Ignored															0	PDE: not present
XD	Reserved		Address of 4KB page frame				Ign.	G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
	Ignored															0	PTE: not present

Trick for XO memory for VMs

EPT usually 1:1 mapping, but we can map it twice

- Once with RWX permissions
- Once with XO permissions

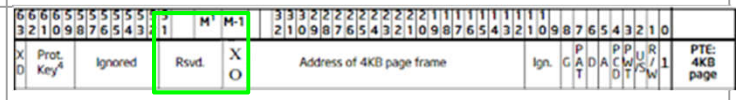
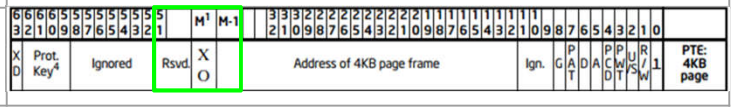
Now guest can switch virtual memory to XO by pointing it to the XO alias of the physical memory



Trick for XO memory for VMs (2)

CPUID Leaf 0x80000008 tells OS how many physical address bits are supported

- When enabled, VMM reduces physical address bits exposed to the host by 1
- KVM CPUID leaf bit tells guest, XO is supported
- Now top physical address bit is an “XO” permission bit

CPUID Leaf 0x80000008	PTE Bit Meaning
Host	
Host - 1	

Qemu/KVM implementation

Pretty small changes for Qemu/KVM

Other KVM based VMMs should have most of the work done for them

Userspace XO support

PROT_EXEC && !PROT_READ = execute only

- Some arm64, pkeys today

Android usage today

Large amount of CPUs out there that already support this, but not enabled

```
/* description of effects of mapping type and prot in current implementation.
* this is due to the limited x86 page protection hardware.  The expected
* behavior is in parens:
*
* map_type     prot
*              PROT_NONE       PROT_READ       PROT_WRITE      PROT_EXEC
* MAP_SHARED   r: (no) no      r: (yes) yes    r: (no) yes     r: (no) yes
*              w: (no) no      w: (no) no      w: (yes) yes    w: (no) no
*              x: (no) no      x: (no) yes     x: (no) yes     x: (yes) yes
*
* MAP_PRIVATE  r: (no) no      r: (yes) yes    r: (no) yes     r: (no) yes
*              w: (no) no      w: (no) no      w: (copy) copy  w: (no) no
*              x: (no) no      x: (no) yes     x: (no) yes     x: (yes) yes
*
* On arm64, PROT_EXEC has the following behaviour for both MAP_SHARED and
* MAP_PRIVATE:
*                                                              r: (no) no
*                                                              w: (no) no
*                                                              x: (yes) yes
*/
```

Running the Kernel in XO

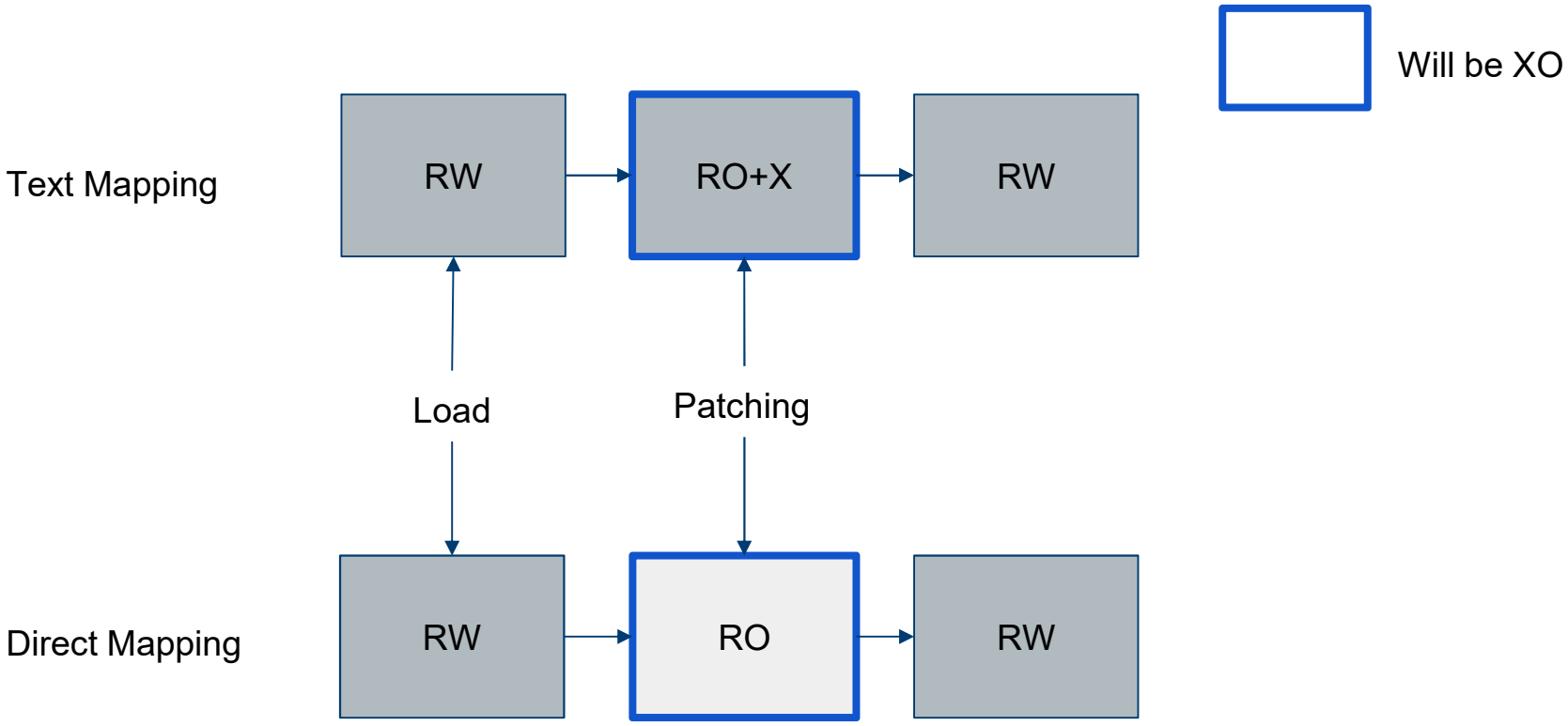
_PAGE_NR

set_memory_nr()

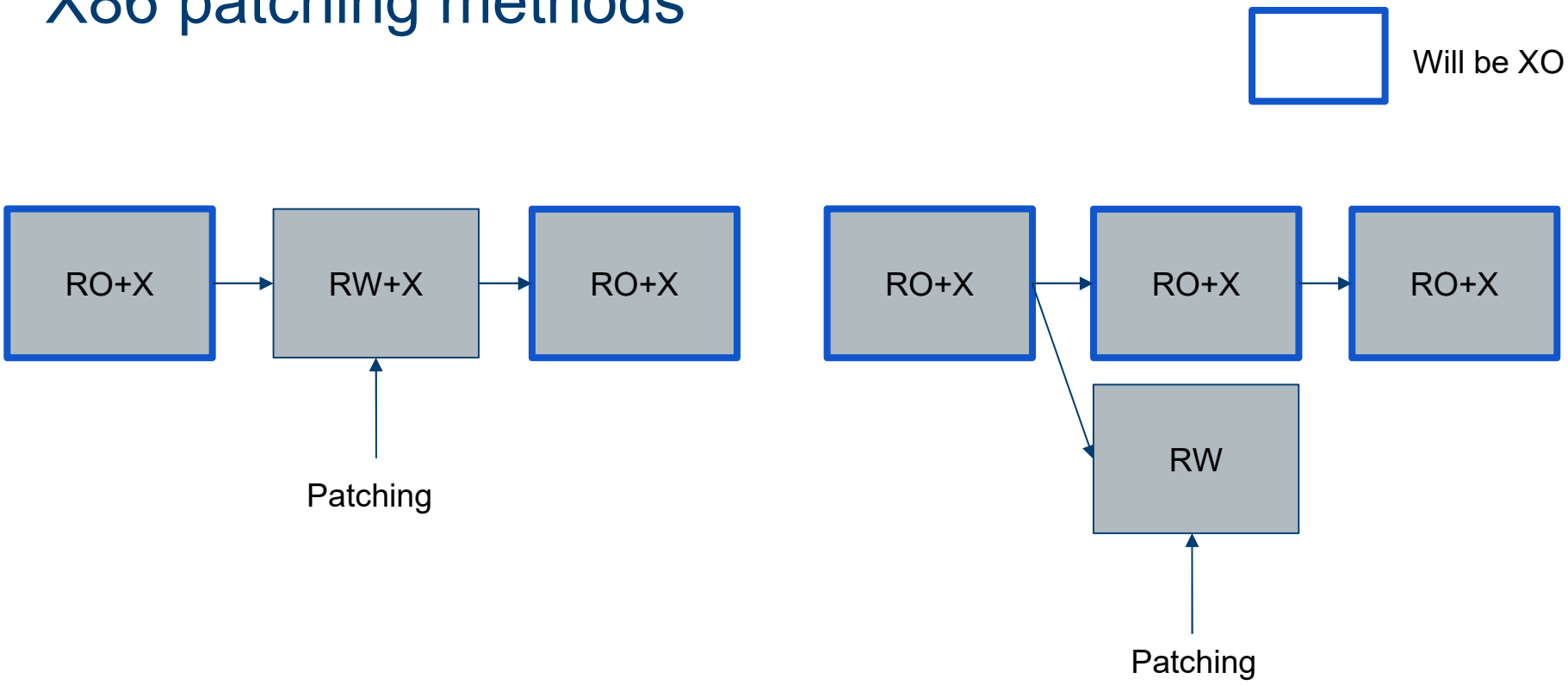
set_memory_r()

Set where appropriate

X86 Kernel Text Permission Lifecycle

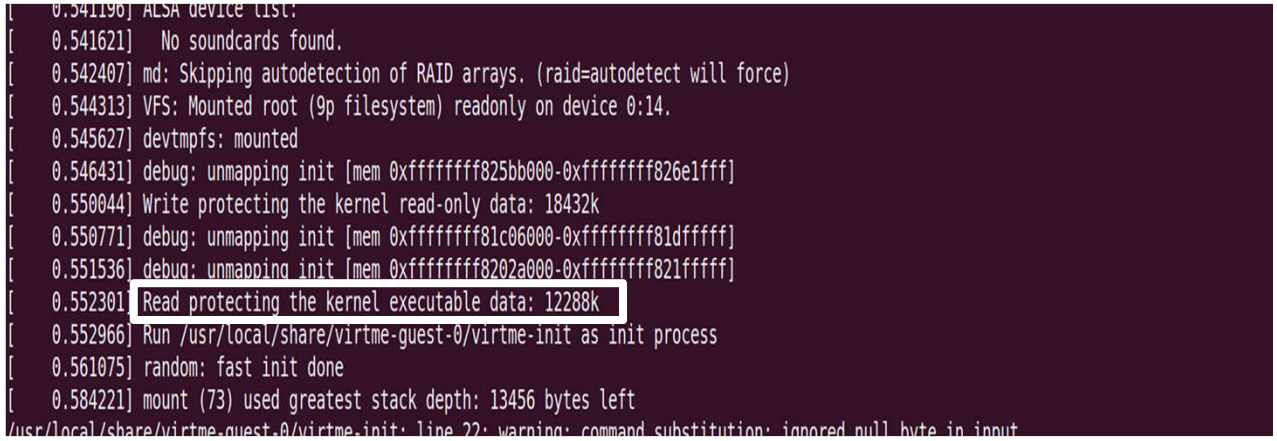


X86 patching methods



So what broke?

Less than expected



- **Text patching features**
- RO data shares page with end of executable data
- Hibernate
- Part of oops message
- Jump tables, literal pools?

Text patching features

- Need to read text to decode old instruction
- We need a kernel text read helper like text_poke()
- For non-xo this will just be a memcpy

Toolchain Mixing Data and Code

Gcc compiler hasn't embedded data in text for long time, if it does should be a bug.

- Separate: iTLB/dTLB, iL1/dL1
- Security purpose, don't mark data as executable, reduce gadgets

Kernel build puts data in text "section", but sets permissions based on _etext symbol.

Performance

Potential performance impact areas

- Very small amount of extra cache pressure from extra EPT pages
- Extra mid-level translation cache pressure
- EPT Memory usage

Didn't expect any significant performance regression

- ~0%, within noise

	Kcbench (higher better)
Normal	3476
XO	3490

Part 3: Making It Reliable

Making this reliable

“IT IS NOT ACCEPTABLE when security people set magical new rules, and then make the kernel panic when those new rules are violated”

What happens if some undiscovered read of kernel text causes the kernel to crash

Two modes

- Strict
- Non-strict

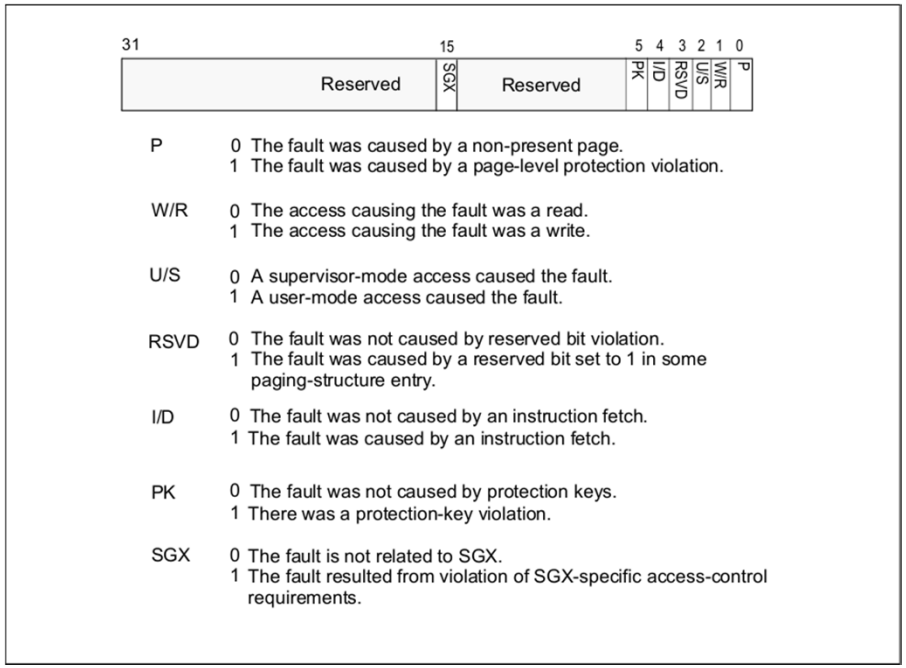
XO faults

KVM injects XO fault with error code:

- P = 1
- W/R = 0
- RSVD = 0
- I/D = 0
- PK = 0
- SGX = 0

Alternative: #VE

- Currently not supported in Kernel, Intel feature to deliver EPT faults directly to guest



P	0	The fault was caused by a non-present page.
	1	The fault was caused by a page-level protection violation.
W/R	0	The access causing the fault was a read.
	1	The access causing the fault was a write.
U/S	0	A supervisor-mode access caused the fault.
	1	A user-mode access caused the fault.
RSVD	0	The fault was not caused by reserved bit violation.
	1	The fault was caused by a reserved bit set to 1 in some paging-structure entry.
I/D	0	The fault was not caused by an instruction fetch.
	1	The fault was caused by an instruction fetch.
PK	0	The fault was not caused by protection keys.
	1	There was a protection-key violation.
SGX	0	The fault is not related to SGX.
	1	The fault resulted from violation of SGX-specific access-control requirements.

Implementing non-strict mode

- Turn off XO protection, log and resume execution
- Potential solutions
  - Change EPT physical permission for faulting page
    - Muddies virtual memory abstraction and creates non-deterministic behavior when physical pages are reused
  - Disable XO for whole system
    - Reduces security benefit
  - Fix guest page tables
    - XO faults in interrupts can race other page table changes

Fixing guest page tables

- XO faults may trigger in an interrupt
  - Need to locklessly change page tables in fault handler
- Wherever an XO fault could happen, need to avoid races by forbidding
  - the page changing permissions
  - breaking a large page
- When changing permissions, need to make sure mapping won't be touched
- Have a POC to avoid these races, but needs more scrutiny

Future - Not reading text as a new rule in the kernel?

In order for XO kernel text to have a future, we need to have a new assumption that the kernel text may not be readable.

Non-XO arch specific code that reads text -- OK!

Modules that need to read themselves for weird reasons -- OK!

- Module param to mark module as not XO compatible

Core code that unconditionally reads text in the core kernel -- NOT OK!

XO arch code that reads text -- NOT OK!

Plans

Infrastructure:

- Land support for userspace XO in Guest kernel/KVM/QEMU
- Land support for Kernel XO
- Watch and wait to see if anything comes up on non-strict mode

Strengthening:

- Protect symtables
- Turn on for BPF JIT?

Summary

We can block common method of working around secret executable code

Cheap perf wise and can be run more safely

Can turn on for large amount of existing HW

Need some cooperation to no longer expect to be able to read executable code

Code

Linux:	https://github.com/redgecombe/linux
KVM:	https://github.com/redgecombe/kvm
Qemu:	https://github.com/redgecombe/qemu

Questions?