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

Rick Edgecombe
Intel provides these materials as-is, with no express or implied warranties.

All products, dates, and figures specified are preliminary, based on current expectations, and are subject to change without notice.

Intel, processors, chipsets, and desktop boards may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Intel technologies’ features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. No product or component can be absolutely secure. Check with your system manufacturer or retailer or learn more at http://intel.com.

Some results have been estimated or simulated using internal Intel analysis or architecture simulation or modeling, and provided to you for informational purposes. Any differences in your system hardware, software or configuration may affect your actual performance.

Intel and the Intel logo are trademarks of Intel Corporation in the United States and other countries.

*Other names and brands may be claimed as the property of others.

© Intel Corporation
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]

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

```c
/*
 * 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?

- Have Arbitrary Call
- Leak text address
- Read rest of text
- ROP to disable SMEP and call userspace
- CRA write gadgets unknown
- Disable userspace exec protections (SMEP)
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
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
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

Address: 0XXXXXXXXXX

Normal

Address: 0XXXXXXXXXX
Address: 1XXXXXXXXXX

XO Enabled

<table>
<thead>
<tr>
<th>RWX GPA EPT</th>
<th>HPA</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>RWX GPA EPT</th>
<th>XO GPA EPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>HPA</td>
<td></td>
</tr>
</tbody>
</table>

Intel
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
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

```c
/* 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:  
   */

struct mapping {
    int map_type;
    int prot;

    int MAP_SHARED  : PROT_NONE   : PROT_READ   : PROT_WRITE  : PROT_EXEC
    + r: (yes) yes    r: (yes) yes    r: (yes) yes    r: (yes) yes
    + w: (no) no      w: (no) no      w: (yes) yes    w: (no) no
    + x: (no) yes     x: (no) yes     x: (yes) yes    x: (yes) yes

    int MAP_PRIVATE : PROT_NONE   : PROT_READ   : PROT_WRITE  : PROT_EXEC
    + r: (yes) yes    r: (yes) yes    r: (no) yes    r: (yes) yes
    + w: (no) no      w: (no) no      w: (copy) copy w: (no) no
    + x: (no) yes     x: (no) yes     x: (yes) 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

Text Mapping

Direct Mapping

RW → RO+X → RW

Load

Patching

RW → RO → RW

Will be XO
X86 patching methods

RO+X → RW+X → RO+X

RO+X → RO+X → RO+X

RW

Patching

Will be XO
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

<table>
<thead>
<tr>
<th></th>
<th>Kcbench (higher better)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>3476</td>
</tr>
<tr>
<td>XO</td>
<td>3490</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Reserved</td>
</tr>
<tr>
<td>15</td>
<td>Reserved</td>
</tr>
<tr>
<td>11</td>
<td>$P$ The fault was caused by a non-present page.</td>
</tr>
<tr>
<td>10</td>
<td>0 The fault was caused by a page-level protection violation.</td>
</tr>
<tr>
<td>11</td>
<td>1 The fault was caused by a page-level protection violation.</td>
</tr>
<tr>
<td>10</td>
<td>$W/R$ The access causing the fault was a read.</td>
</tr>
<tr>
<td>11</td>
<td>0 The access causing the fault was a write.</td>
</tr>
<tr>
<td>10</td>
<td>1 The access causing the fault was a write.</td>
</tr>
<tr>
<td>9</td>
<td>$U/S$ A supervisor-mode access caused the fault.</td>
</tr>
<tr>
<td>8</td>
<td>0 A user-mode access caused the fault.</td>
</tr>
<tr>
<td>7</td>
<td>1 A user-mode access caused the fault.</td>
</tr>
<tr>
<td>6</td>
<td>$RSVD$ The fault was not caused by reserved bit violation.</td>
</tr>
<tr>
<td>6</td>
<td>0 The fault was not caused by reserved bit violation.</td>
</tr>
<tr>
<td>5</td>
<td>1 The fault was caused by a reserved bit set to 1 in some paging-structure entry.</td>
</tr>
<tr>
<td>4</td>
<td>$I/D$ The fault was not caused by an instruction fetch.</td>
</tr>
<tr>
<td>3</td>
<td>0 The fault was not caused by an instruction fetch.</td>
</tr>
<tr>
<td>2</td>
<td>1 The fault was caused by an instruction fetch.</td>
</tr>
<tr>
<td>1</td>
<td>$PK$ The fault was not caused by protection keys.</td>
</tr>
<tr>
<td>0</td>
<td>0 The fault was not caused by protection keys.</td>
</tr>
<tr>
<td></td>
<td>1 There was a protection-key violation.</td>
</tr>
<tr>
<td>7</td>
<td>$SGX$ The fault is not related to SGX.</td>
</tr>
<tr>
<td>6</td>
<td>0 The fault is not related to SGX.</td>
</tr>
<tr>
<td>5</td>
<td>1 The fault resulted from violation of SGX-specific access-control requirements.</td>
</tr>
</tbody>
</table>
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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux:</td>
<td><a href="https://github.com/redgecombe/linux">https://github.com/redgecombe/linux</a></td>
</tr>
<tr>
<td>KVM:</td>
<td><a href="https://github.com/redgecombe/kvm">https://github.com/redgecombe/kvm</a></td>
</tr>
<tr>
<td>Qemu:</td>
<td><a href="https://github.com/redgecombe/qemu">https://github.com/redgecombe/qemu</a></td>
</tr>
</tbody>
</table>
Questions?