INTEL OPEN SOURCE TECHNOLOGY CENTER | 01.org

Proactive Defense Against CPU Side Channel Attacks

Kristen Carlson Accardi

“a **monoculture** is a community of computers that all run identical software. All the computer systems in the community thus have the same vulnerabilities, and, like agricultural monocultures, are subject to catastrophic failure in the event of a successful attack.”

https://en.wikipedia.org/wiki/Monoculture_(computer_science)

What does this have to do with side channels?

Software Diversification makes some side channels less useful

2014 IEEE Symposium on Security and Privacy

SoK: Automated Software Diversity

Per Larsen, Andrei Homescu, Stefan Brunthaler, Michael Franz
University of California, Irvine

Gadge Me If You Can

Secure and Efficient Ad-hoc Instruction-Level Randomization for x86 and ARM

Lucas Davi[1,2], Alexandra Dmitrienko[3], Stefan Nürnberger[2], Ahmad-Reza Sadeghi[1,2,3]

Abstr
wo deca
and the
tore" m
he inter
iterature
2008.
[redacted]

Mauro Conti, Stephen Crane, Tommaso Frassetto, Andrei Homescu, Georg Koppen, Per Larsen,
Christopher Liebchen, Mike Perry, and

Selfrando: Securing
De-anonymization

Abstract: Tor is a well-known anonymous c
tion system used by millions of users, incl

Compiler-assisted Code Randomization

Hyungjoon Koo,* Yaohui Chen,† Long Lu,† Vasileios P. Kemerlis,‡ Michalis Polychronakis*

*Stony Brook University {hykoo, mikepo}@cs.stonybrook.edu
†Northeastern University {yaohway, long}@ccs.neu.edu
‡Brown University vpk@cs.brown.edu

of research on software diversi-
e layout randomization has seen
randomization, an effective defense

under certain circumstances by remotely leaking [18–20] or inferring [21,22] what code exists at a given memory location. As a response, recent protections against JIT-ROP exploits rely

kR^X: Comprehensive Kernel Protection against Just-In-Time Code Reus

Marios Pomonis* Theofilos Petsios* Angelos D. Ke
Michalis Polychronakis† Vasileios P. Kemerlis†
*Columbia University {mpomonis, theofilos, angelos}@cs.columbia.edu
†Stony Brook University mikepo@cs.stonybrook.edu

Abstract

The abundance of memory corruption and disclosure vulnerabilities in kernel code necessitates the deployment of hardening techniques to prevent privilege escalation attacks.

1. Introduction

The deployment of standar
address space layout rando
executable memory [71], I

Operating System Protection Through Program Evolution

by Dr. Frederick B. Cohen ‡

Simplified Elf Executable format

Elf Header
Segment Table
.text
.data
.rodata
.bss
Symbol table
String tables

- Kernel is much more complicated
- .text is your executable code

Levels of Randomization

- Instruction level
  - Equivalent instruction substitution
  - Instruction reordering
  - Register allocation reordering
  - Garbage code insertion (nops etc)
- Basic block
  - Reordering
- Function level
  - Stack layout randomization
    - -fstack-shuffle (OpenBSD)
  - Function parameter randomization
  - Inlining, outlining or splitting
  - Jump table insertion
- Program level
  - Function reordering
  - Base Address Randomization
  - Program encoding
  - Data randomization
    - Static data layout
    - Constant blinding
    - Structure layout randomization
    - Heap/stack layout randomization
  - System call mapping randomization

When to Randomize

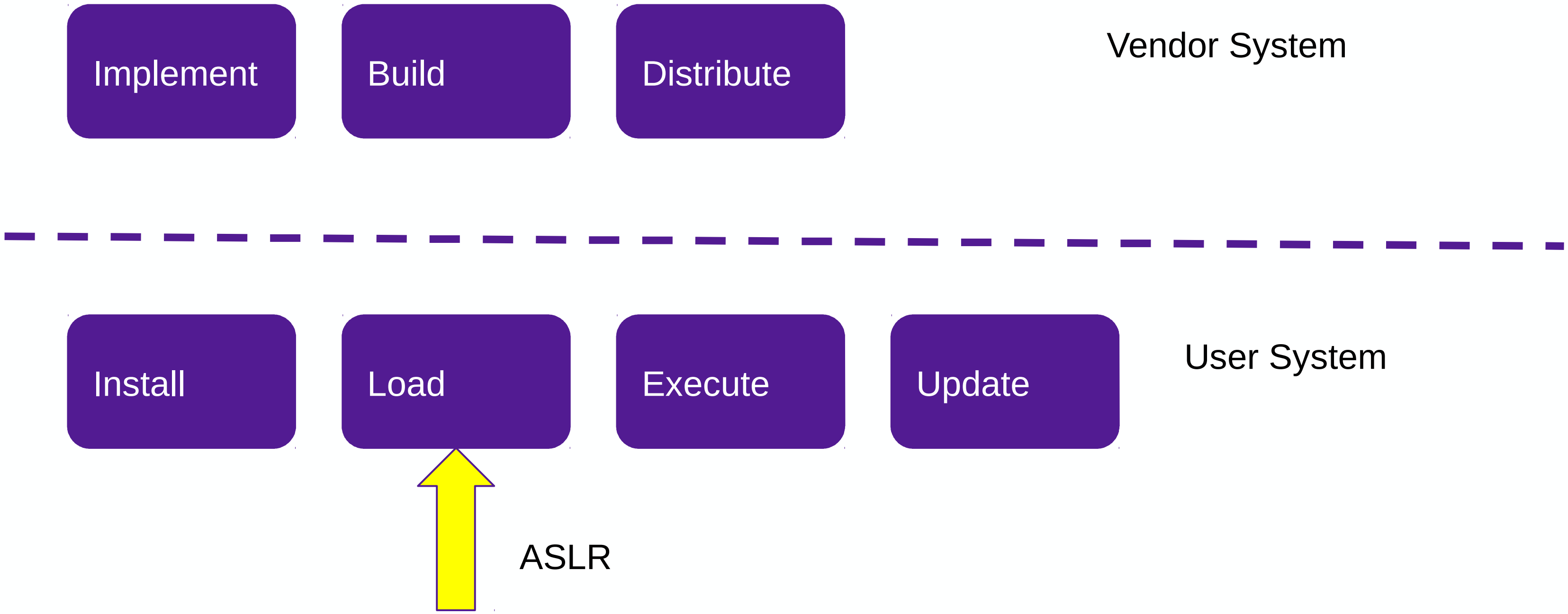
Implement | Build | Distribute — Vendor System

Install | Load | Execute | Update — User System

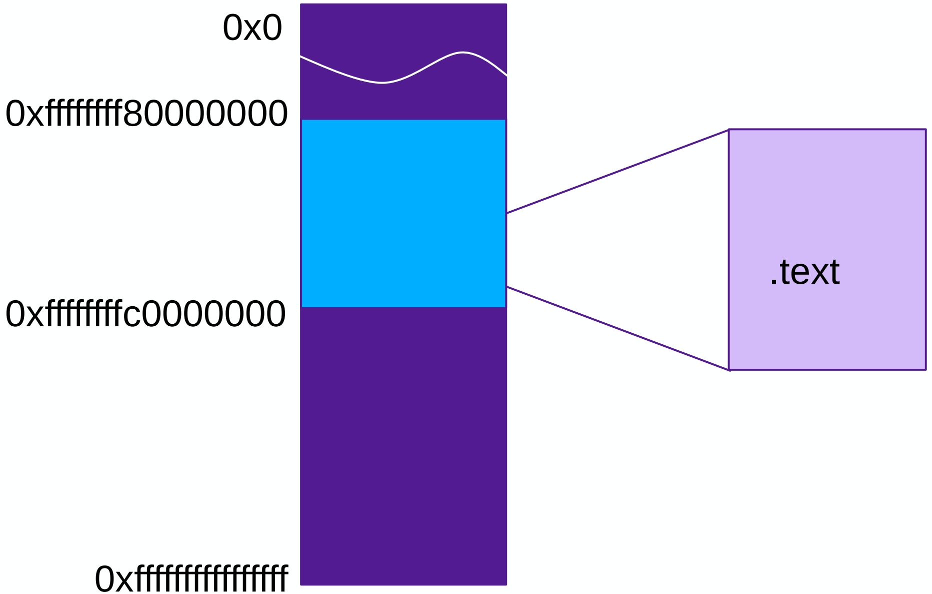
Levels of Randomization

- Instruction level
  - Equivalent instruction substitution
  - Instruction reordering
  - Register allocation reordering
  - Garbage code insertion (nops etc)
- Basic block
  - Reordering
- Function level
  - Stack layout randomization
    - -fstack-shuffle (OpenBSD)
  - Function parameter randomization
  - Inlining, outlining or splitting
  - Jump table insertion
- Program level
  - Function reordering
  - Base address randomization
    - KASLR
  - Program encoding
  - Data randomization
    - Static data layout
    - Constant blinding
    - Structure layout randomization
      - Gcc plugin currently implemented in kernel
    - Heap/stack layout randomization
  - System call mapping randomization

When to Randomize



KASLR (Kernel Address Space Layout Randomization)



- Kernel Text section is located within a fixed range
- Exact location is determined at boot
- Order within .text is unchanged

ASLR is pretty weak

- Kernel ASLR has low entropy
  - Brute Force attacks are possible
- Infoleaks reveal location of entire .text segment
  - Relative distances remain the same

If ASLR is so weak, why don’t we do more?

Monoculture has its benefits

- Ease of Distribution
  - Creating randomized binaries at download time is slow and expensive
  - Would only work for people who create custom kernels
- Code signing
- Load/Run time overhead of diversified binaries
- Debugging & error reporting
- Tracing and live-patching

Can we do better?

When to Randomize

Implement | Build | Distribute

Vendor System

Install | Load | Execute | Update

User System

Levels of Randomization

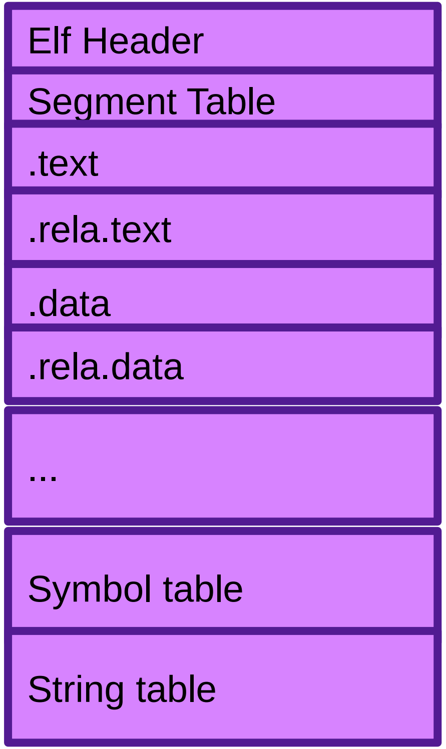
- Instruction level
  - Equivalent instruction substitution
  - Instruction reordering
  - Register allocation reordering
  - Garbage code insertion (nops etc)
- Basic block
  - Reordering
- Function level
  - Stack layout randomization
    - -fstack-shuffle (OpenBSD)
  - Function parameter randomization
  - Inlining, outlining or splitting
  - Jump table insertion
- Program level
  - Function reordering
  - Base address randomization
    - KASLR
  - Program encoding
  - Data randomization
    - Static data layout
    - Constant blinding
    - Structure layout randomization
      - Gcc plugin currently implemented in kernel
    - Heap/stack layout randomization
  - System call mapping randomization

Implementing kernel function reordering

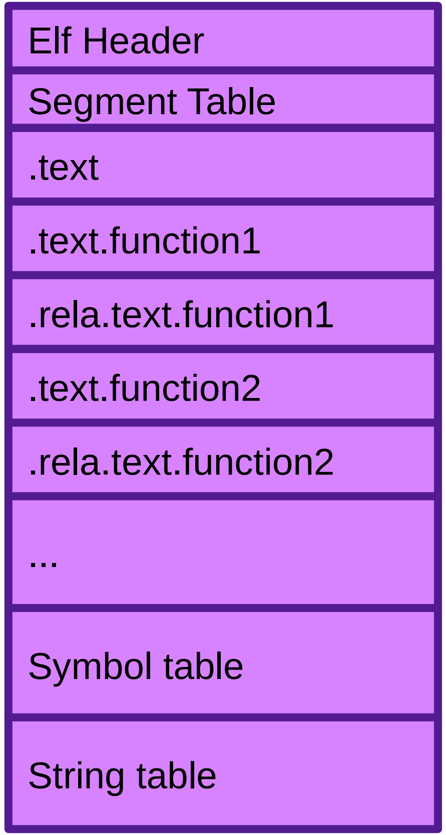
- Keep it small and simple by starting with modules
- Modules are already linked into the kernel at module load time
- Modules are already relocatable objects
- Modules can have individual Makefiles

```
obj-$(CONFIG_TEST_MODULE) +=
test_module.o
ccflags-y := -ffunction-sections
```

Relocatable format



-ffunction-sections



Simple module example

```
static void __attribute__((optimize("O0"))) test_module_do_work(void)
{
	...
}

static void test_module_wq_func(struct work_struct *w)
{
	test_module_do_work();
	queue_work(test_module_wq, w);
	return;
}

static int __init test_module_init(void)
{
	...
}
```

Implementing kernel function reordering

```
[kcaccard@kcaccard-mobl3 test_module]$ readelf --sections --wide test_module.ko


[Nr] Name                          Type      Address            Off     Size    ES Flg Lk Inf Al
[ 2] .text                         PROGBITS 0000000000000000 000064 000000 00   AX  0
0  1
[ 3] .text.test_module_do_work PROGBITS 0000000000000000 000064 000033 00  AX  0   0
1
[ 5] .text.test_module_wq_func PROGBITS 0000000000000000 000097 000023 00  AX  0   0
1
```

Implementing kernel function reordering

```
[kcaccard@kcaccard-mobl3 test_module]$ readelf --sections --wide test_module.ko

[Nr] Name                    Type      Address          Off    Size   ES Flg Lk Inf Al
[ 2] .text                       PROGBITS 0000000000000000 000064 000000 00  AX  0
0  1
[ 3] .text.test_module_do_work PROGBITS 0000000000000000 000064 000033 00  AX  0   0
1
[ 5] .text.test_module_wq_func PROGBITS 0000000000000000 000097 000023 00  AX  0   0
1


[kcaccard@kcaccard-mobl3 test_module]$ readelf --relocs --wide test_module.ko

Relocation section '.rela.text.test_module_wq_func' at offset 0x1d830 contains 4
entries:
Offset            Info             Type            Symbol's Value  Symbol's Name +
Addend
0000000000000001  0000003200000002 R_X86_64_PC32   0000000000000000 __fentry__ - 4
000000000000000a  0000000300000002 R_X86_64_PC32   0000000000000000
.text.test_module_do_work - 4
0000000000000014  0000000f00000002 R_X86_64_PC32   0000000000000000 .bss - 4
000000000000001f  0000003000000002 R_X86_64_PC32   0000000000000000 queue_work_on - 4
```

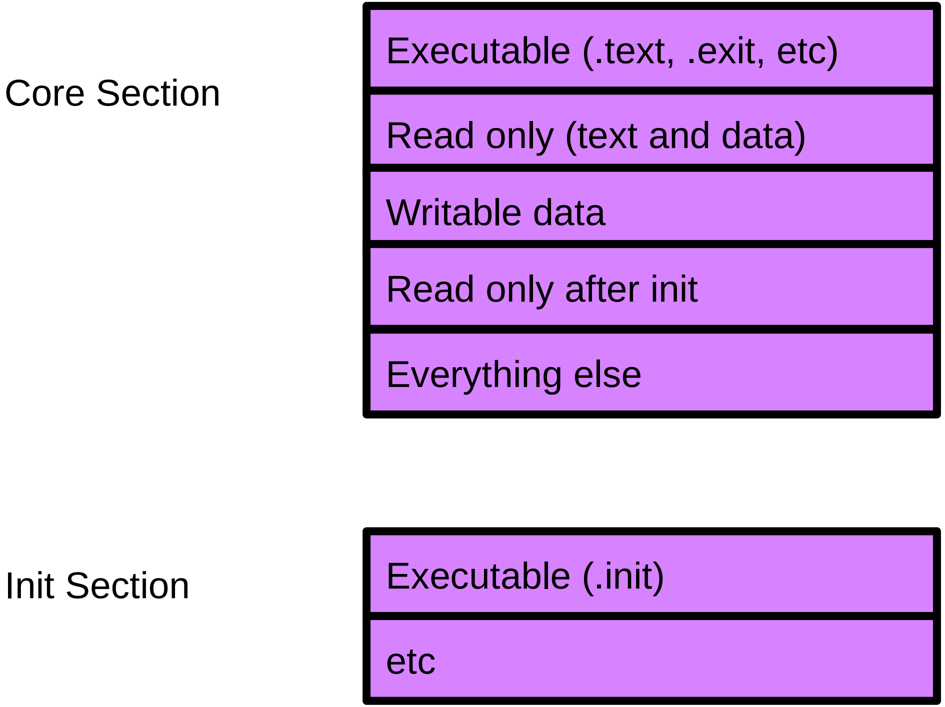
Implementing kernel function reordering

```
[kcaccard@kcaccard-mobl3 test_module]$ readelf --relocs --wide test_module.ko
Relocation section '.rela.text.test_module_wq_func' at offset 0x1d830 contains 4
entries:
Offset              Info             Type            Symbol's Value  Symbol's Name +
Addend
000000000000000a  0000000300000002 R_X86_64_PC32   0000000000000000
.text.test_module_do_work - 4

[kcaccard@kcaccard-mobl3 test_module]$ objdump -d test_module.ko
Disassembly of section .text.test_module_wq_func:

0000000000000000 <test_module_wq_func>:
   0:	e8 00 00 00 00       	callq  5 <test_module_wq_func+0x5>
   5:	53                   	push   %rbx
   6:	48 89 fb             	mov    %rdi,%rbx
   9:	e8 00 00 00 00       	callq  e <test_module_wq_func+0xe>
   e:	48 89 da             	mov    %rbx,%rdx
  11:	48 8b 35 00 00 00 00 	mov    0x0(%rip),%rsi        # 18
<test_module_wq_func+0x18>
  18:	bf 40 00 00 00       	mov    $0x40,%edi
  1d:	5b                   	pop    %rbx
  1e:	e9 00 00 00 00       	jmpq   23 <work+0x3>
```

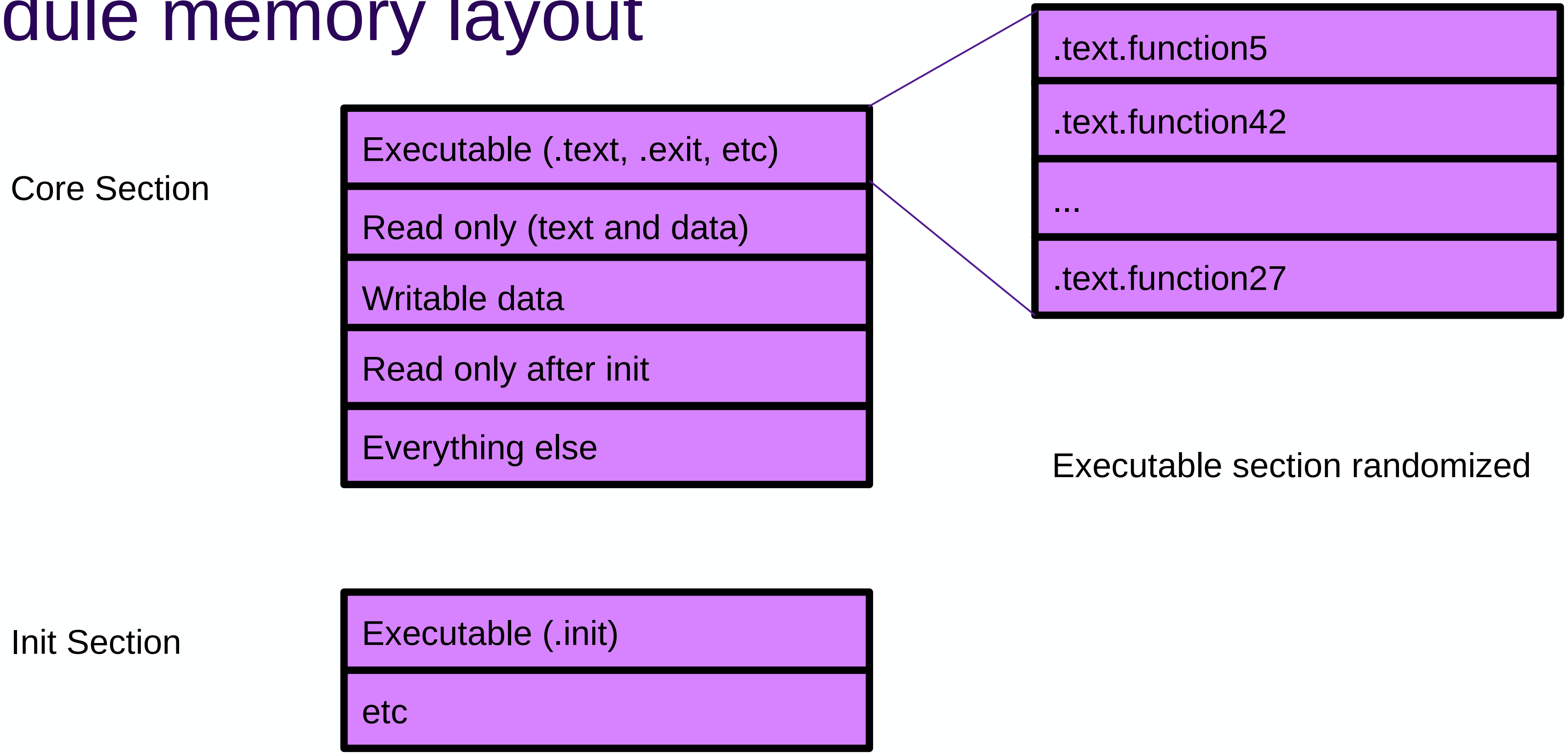
Implementing kernel function reordering

```
[kcaccard@kcaccard-mobl3 test_module]$ readelf --symbols --wide test_module.ko

Symbol table '.symtab' contains 58 entries:
   Num: Value            Size    Type     Bind   Vis      Ndx Name
    34: 0000000000000000    51 FUNC    LOCAL  DEFAULT    3 test_module_do_work
    36: 0000000000000000    35 FUNC    LOCAL  DEFAULT    5 test_module_wq_func
    37: 0000000000000000   214 FUNC    LOCAL  DEFAULT    7 test_module_init
    38: 0000000000000000    36 FUNC    LOCAL  DEFAULT    9 test_module_exit
    49: 0000000000000000    36 FUNC    GLOBAL DEFAULT    9 cleanup_module
    51: 0000000000000000   214 FUNC    GLOBAL DEFAULT    7 init_module
```

Module memory layout



Module memory layout



Effectiveness

- Depends on the number of functions
  - Inherently not as strong as even finer grained randomization
  - Can expand over time
- Randomizing modules isn't the end goal
  - Will need some modifications to implementation for kernel .text
  - Implementing for modules lets us take incremental steps

Monoculture benefits revisited

- Ease of Distribution ✔
- Code signing ✔
- Load/Run time overhead of diversified binaries ?
- Debugging & error reporting ?
- Tracing and live-patching ✔ ?

We can do better still

Execute Only Memory

- x86 page tables have bits for Present, Write, and No-eXecute (NX).
- Present must be set for all Write and NX entries. It is not possible to represent a Writeable, but unreadable or executable but unreadable page table entry.
- Extended Page Table (EPT, part of VMX) format contains separate Read, Write and eXecute bits. It can represent Present+eXecutable+non-Readable memory.
  - Function pointers in data would still be readable

- Implementing this requires work both in the guest OS and the VMM.
- Challenges are still unknown
  - Data in text area may still exist (jump tables)
  - May need to be turned on and off for some reason (kprobes)

Just a few resources

- https://www3.cs.stonybrook.edu/~mikepo/papers/krx.eurosys17.pdf
- https://www.ics.uci.edu/~perl/pets16_selfrando.pdf
- https://www.computer.org/csdl/proceedings/sp/2014/4686/00/4686a276.pdf
- https://www.ics.uci.edu/~perl/oakland15_readactor.pdf
- https://cs.brown.edu/~vpk/papers/ccr.sp18.pdf

POC available at:
https://github.com/kaccardi/linux.git (reorder-module-functions branch)

INTEL OPEN SOURCE TECHNOLOGY CENTER | 01.org

@kcaccardi