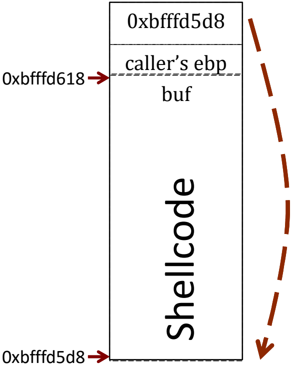
Derandomizing Kernel Address Space Layout for Memory Introspection and Forensics

Yufei Gu, and **Zhiqiang Lin**

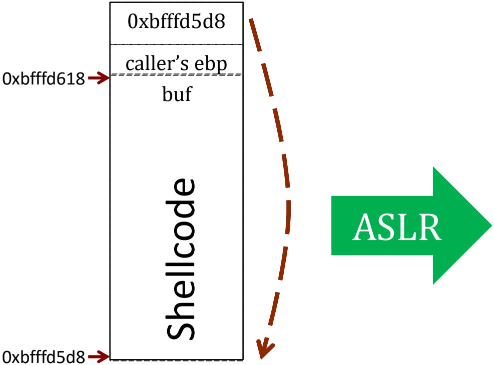
The University of Texas at Dallas

March 9^{th}, 2016

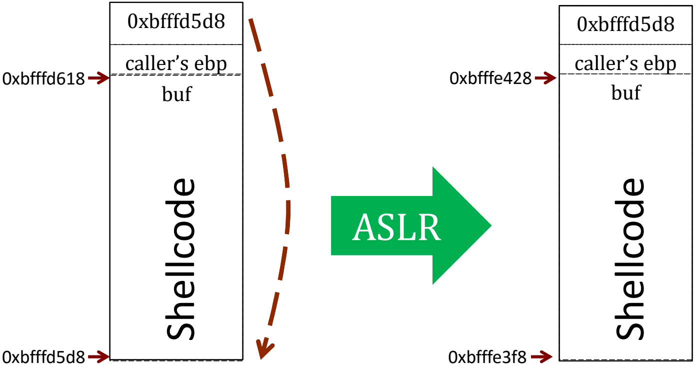
What is ASLR [Tea00]



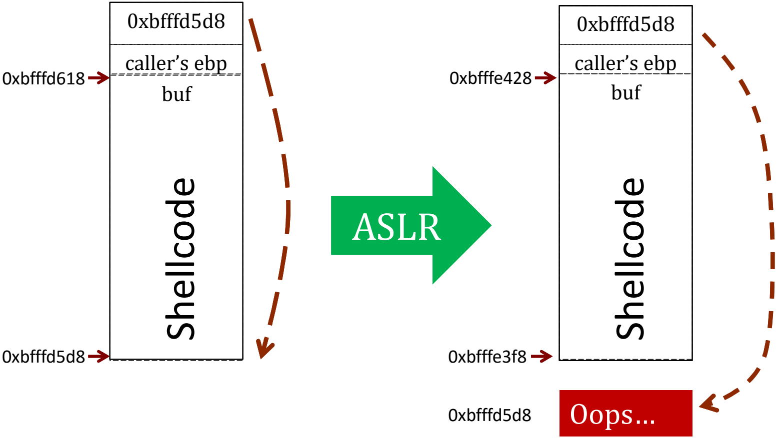
What is ASLR [Tea00]



What is ASLR [Tea00]



What is ASLR [Tea00]



Why Kernel ASLR

- **Kernel exploits**
    - Kernel buffer overflow
    - Kernel ROP [Sha07, BRSS08]
- **Kernel rootkits**
    - Tampering with the same virtual address

Why Kernel ASLR

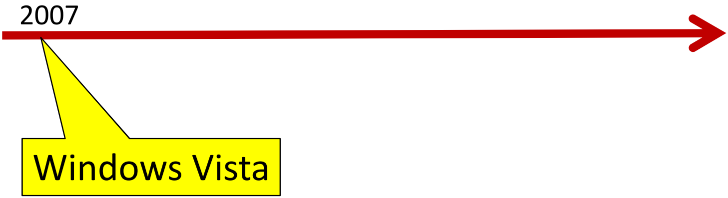
- **Kernel exploits**
  - Kernel buffer overflow
  - Kernel ROP [Sha07, BRSS08]
- **Kernel rootkits**
  - Tampering with the same virtual address

Modern OS kernels including Windows, Linux, and Mac OS all have adopted kernel ASLR

Kernel ASLR

2007

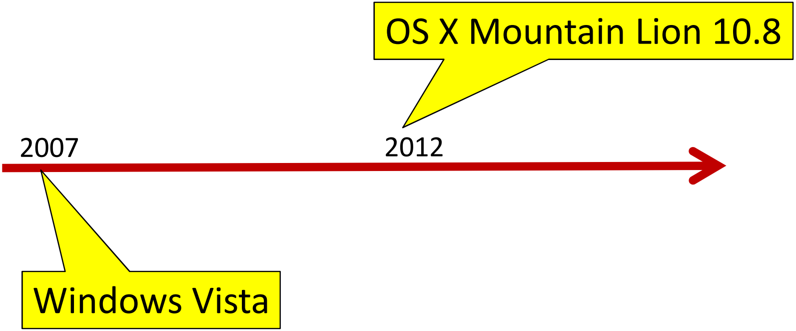
Kernel ASLR



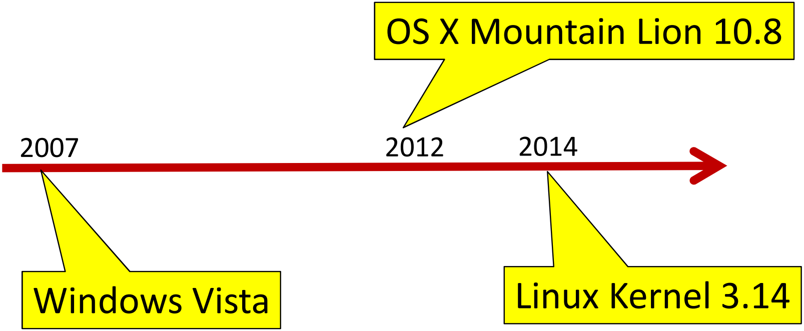
2007

Windows Vista

Kernel ASLR



Kernel ASLR



OS X Mountain Lion 10.8

2007

2012

2014

Windows Vista

Linux Kernel 3.14

Consequences of Kernel ASLR

It significantly **decreases the success rate** of kernel memory **exploits** as well as some kernel **rootkit** attacks

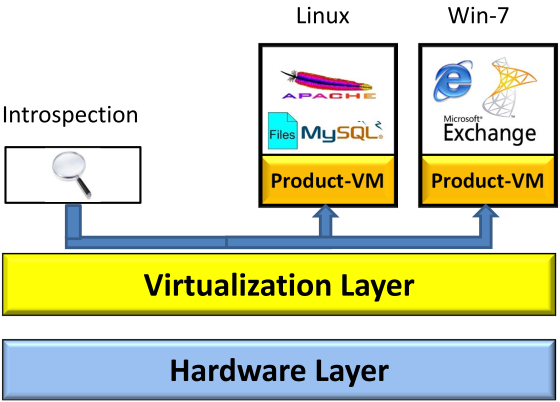
Consequences of Kernel ASLR

It significantly **decreases the success rate** of kernel memory **exploits** as well as some kernel **rootkit** attacks

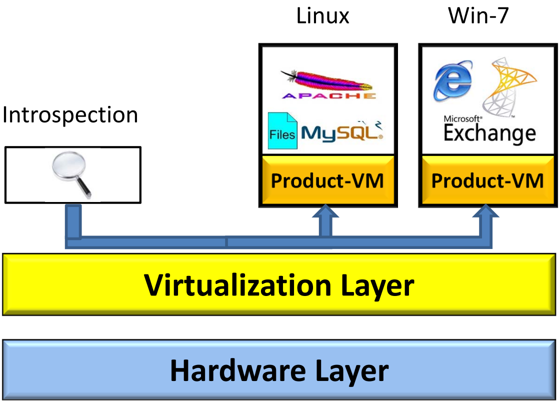
It also **hinders** the applications of

1. **Kernel introspection** [GR03]
2. **Kernel memory forensics** [Wal05]

Introspection [GR03] and Memory Forensics [Wal05]



Introspection [GR03] and Memory Forensics [Wal05]



Linux

Win-7

Introspection

Product-VM

Product-VM

Virtualization Layer

Hardware Layer

Introspection and forensic often need to know **where** kernel code and data is located

Knowning the specific kernel address is important

For an instrospection tool:

1. To **interpret** a system call event, it requires to know the address of the system call tables (e.g., [FLH13])
2. To **intercept** the kernel object allocation and deallocation, it requires to know the addresses of the functions that manages the kernel heaps (e.g., [ZL15])
3. To **traverse** certain dynamically allocated kernel objects, it needs to know their rooted global addresses (e.g., [FLB15])

Knowning the specific kernel address is important

For an instrospection tool:

1. To **interpret** a system call event, it requires to know the address of the system call tables (e.g., [FLH13])
2. To **intercept** the kernel object allocation and deallocation, it requires to know the addresses of the functions that manages the kernel heaps (e.g., [ZL15])
3. To **traverse** certain dynamically allocated kernel objects, it needs to know their rooted global addresses (e.g., [FLB15])

For virtual machine introspection and forensics to be effective, we must **derandomize** kernel ASLR

Derandomizing ASLR at User Space

Derandomizing ASLR at User Space

1. **Brute-force linear search** [SPP+04], which only requires 2^{16} probes to derandomize the address space of a vulnerable program for a 32-bit ASLR implementation.

Derandomizing ASLR at User Space

1. **Brute-force linear search** [SPP+04], which only requires 2^{16} probes to derandomize the address space of a vulnerable program for a 32-bit ASLR implementation.
2. **Information leakage** [RMPB09] by exploiting information about the base address of `libc`, also code fragments available at fixed locations to discover the address of `libc` functions.

Derandomizing ASLR at User Space

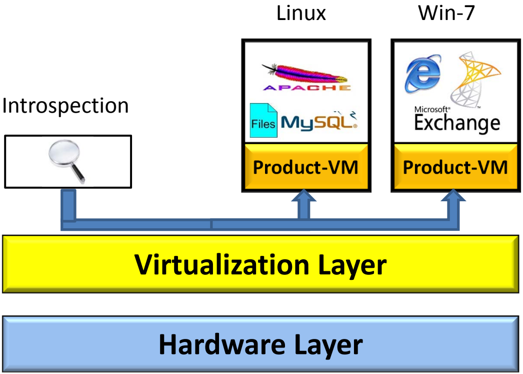
1. **Brute-force linear search** [SPP+04], which only requires 2^{16} probes to derandomize the address space of a vulnerable program for a 32-bit ASLR implementation.
2. **Information leakage** [RMPB09] by exploiting information about the base address of `libc`, also code fragments available at fixed locations to discover the address of `libc` functions.
3. JIT-ROP [SMD+13] attack, which leverages **multiple memory disclosures** to bypass the ASLR

Derandomizing ASLR at User Space

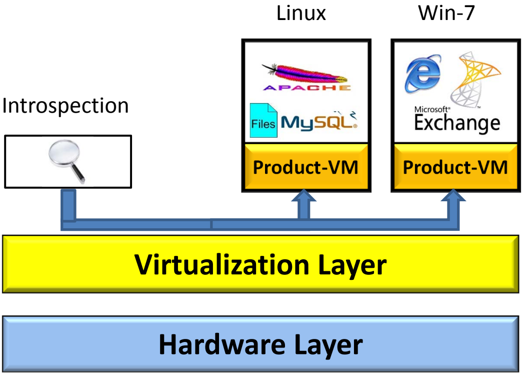
1. **Brute-force linear search** [SPP+04], which only requires 2^{16} probes to derandomize the address space of a vulnerable program for a 32-bit ASLR implementation.
2. **Information leakage** [RMPB09] by exploiting information about the base address of `libc`, also code fragments available at fixed locations to discover the address of `libc` functions.
3. JIT-ROP [SMD+13] attack, which leverages **multiple memory disclosures** to bypass the ASLR

These offensive approaches only have the **remote access** of the target machine

VMI and Forensics Have **Local Access**



VMI and Forensics Have **Local Access**



VMI and forensics applications have the physical access of the target machine

- **CPU registers**
- **Physical memory**

Too many options (e.g., too **many signatures**) for derandomization

Derandomization Kernel ASLR by Volatility [Wal05]

Kernel Version	Signature (Byte Sequence)	Size (Bytes)
VistaSP0x86	00 00 00 00 00 00 00 00 4b 44 42 47 28 03	14
VistaSP1x86	00 00 00 00 00 00 00 00 4b 44 42 47 30 03	14
VistaSP2x86	00 00 00 00 00 00 00 00 4b 44 42 47 30 03	14
VistaSP0x64	00 f8 ff ff 4b 44 42 47 28 03	10
VistaSP1x64	00 f8 ff ff 4b 44 42 47 30 03	10
VistaSP2x64	00 f8 ff ff 4b 44 42 47 30 03	10
Win7SP1x64	00 f8 ff ff 4b 44 42 47 40 03	10
Win7SP1x86	00 00 00 00 00 00 00 00 4b 44 42 47 40 03	14
Win7SP0x86	00 00 00 00 00 00 00 00 4b 44 42 47 40 03	14
Win7SP0x64	00 f8 ff ff 4b 44 42 47 40 03	10
Win2008SP1x86	00 00 00 00 00 00 00 00 4b 44 42 47 30 03	14
Win2008SP2x86	00 00 00 00 00 00 00 00 4b 44 42 47 30 03	14
Win2008SP1x64	00 f8 ff ff 4b 44 42 47 30 03	10
Win2008SP2x64	00 f8 ff ff 4b 44 42 47 30 03	10
Win2008R2SP0x64	00 f8 ff ff 4b 44 42 47 40 03	10
Win2008R2SP1x64	00 f8 ff ff 4b 44 42 47 40 03	10
Win8SP0x86	00 00 00 00 00 00 00 00 4b 44 42 47 60 03	14
Win8SP1x86	00 00 00 00 00 00 00 00 4b 44 42 47 60 03	14
Win8SP0x64	03 f8 ff ff 4b 44 42 47 60 03	10
Win8SP1x64	03 f8 ff ff 4b 44 42 47 60 03	10
Win2012x64	03 f8 ff ff 4b 44 42 47 60 03	10
Win2012R2x64	03 f8 ff ff 4b 44 42 47 60 03	10

Table: **KDBG Signatures** used by Volatility to Derandomize the Kernel.

Problem Statement, Scope, Threat Model

Problem Statement

Investigate the **optimal solutions** for derandomizing the kernel address space for introspection and forensics

1. **Robust**
2. **Efficient**

Problem Statement, Scope, Threat Model

Problem Statement

Investigate the **optimal solutions** for derandomizing the kernel address space for introspection and forensics

1. **Robust**
2. **Efficient**

Scope

We focus on Linux kernel

Problem Statement, Scope, Threat Model

Problem Statement

Investigate the **optimal solutions** for derandomizing the kernel address space for introspection and forensics

1. **Robust**
2. **Efficient**

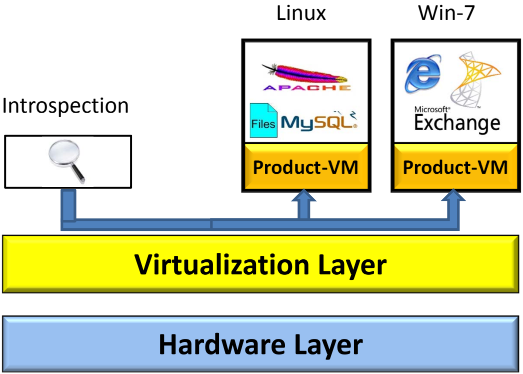
Scope

We focus on Linux kernel

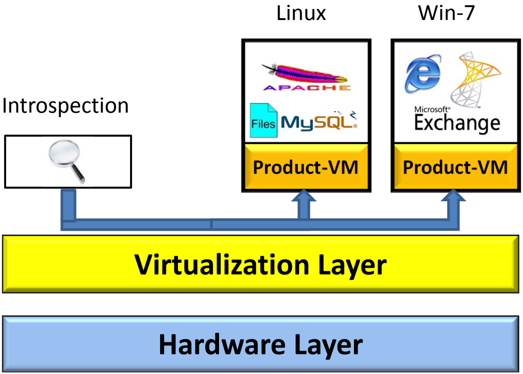
Threat Model

- Non Cooperative OS
- OS has been compromised

Solution Space



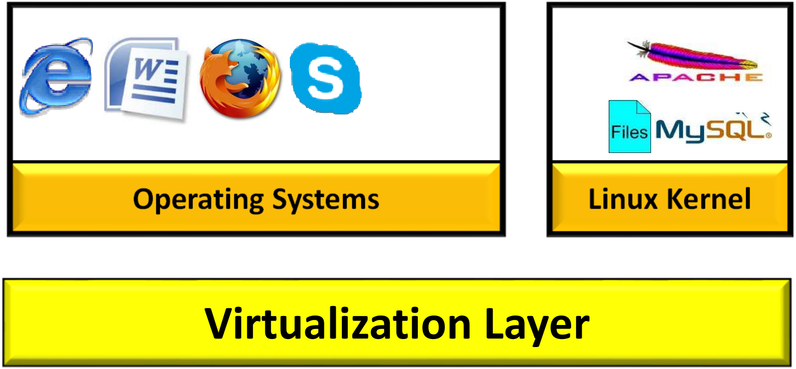
Solution Space



Deriving signatures from

- **Kernel Code**
- **Kernel Data**

Solution Space

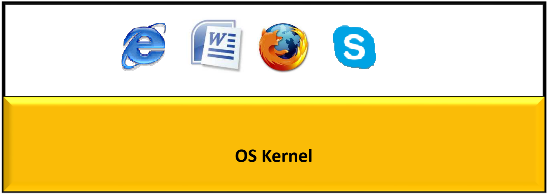


Operating Systems

Linux Kernel

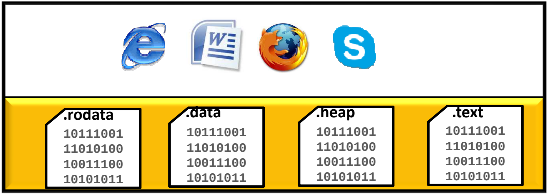
Virtualization Layer

Solution Space

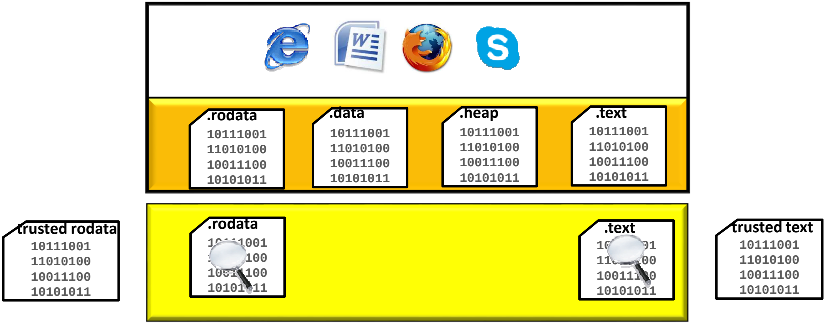


OS Kernel

Solution Space



Solution Space



Challenges from (Modifiable) Kernel Code

1. **Relocation**
2. **Alternative Instructions**
3. **Symmetric Multiprocessing**
4. **Function Tracing**

Relocation

Relocation is typically needed by a linker when linking object code to produce the final executable. Relocation is also needed when loading kernel modules or loading ASLR-enabled kernel.

Example

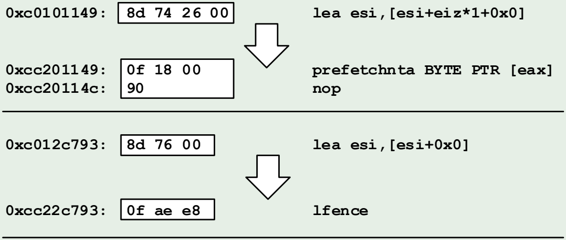
```
0xc0100033: b9 00 b0 a7 00          mov ecx,0xa7b000
                  ⇩
0xcc200033: b9 00 b0 b7 0c          mov ecx,0xcb7b000
------------------------------------------------------------------------
0xc0103045: 89 0c c5 00 a0 9e c0    mov DWORD PTR [eax*8-0x3f616000],ecx
                        ⇩
0xcc203045: 89 0c c5 00 a0 ae cc    mov DWORD PTR [eax*8-0x33516000],ecx
------------------------------------------------------------------------
```

Alternative Instructions

Kerne will dynamically replace some (old) instructions with more efficient alternatives.

Example

```
#define mb() alternative("lock; addl $0,0(%%esp)",
"mfence", X86_FEATURE_XMM2)
```

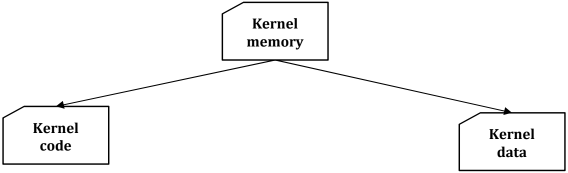


```
0xc0101149: 8d 74 26 00        lea esi,[esi+eiz*1+0x0]

0xcc201149: 0f 18 00           prefetchnta BYTE PTR [eax]
0xcc20114c: 90                 nop

0xc012c793: 8d 76 00           lea esi,[esi+0x0]

0xcc22c793: 0f ae e8           lfence
```

An Overview of the Investigated Approaches

Kernel memory

An Overview of the Investigated Approaches

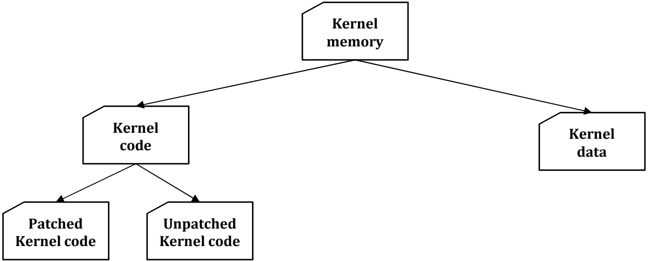


Kernel memory

Kernel code

Kernel data

An Overview of the Investigated Approaches



Kernel memory

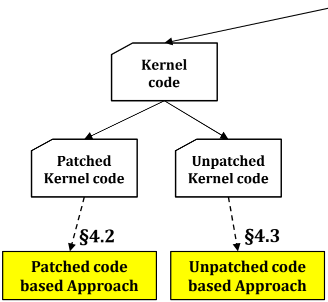
Kernel code

Kernel data

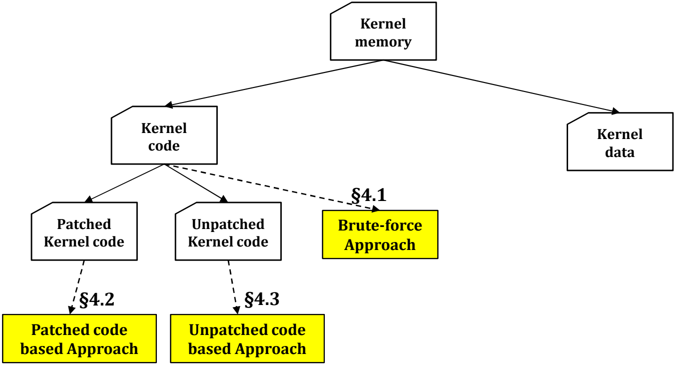
Patched Kernel code

Unpatched Kernel code

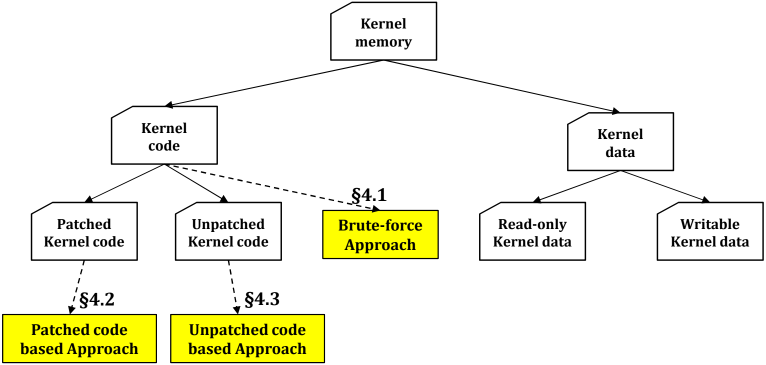
An Overview of the Investigated Approaches



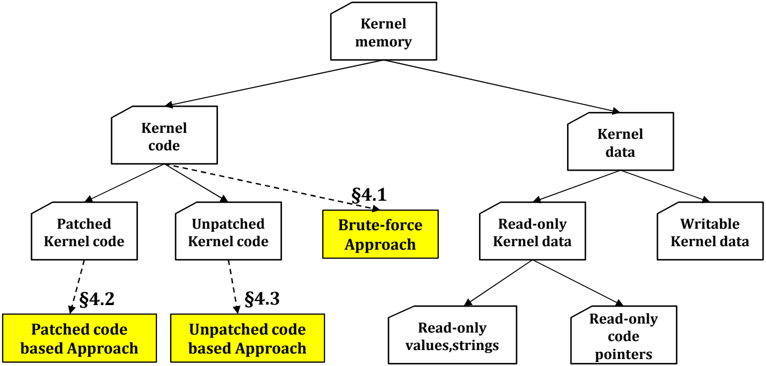
An Overview of the Investigated Approaches



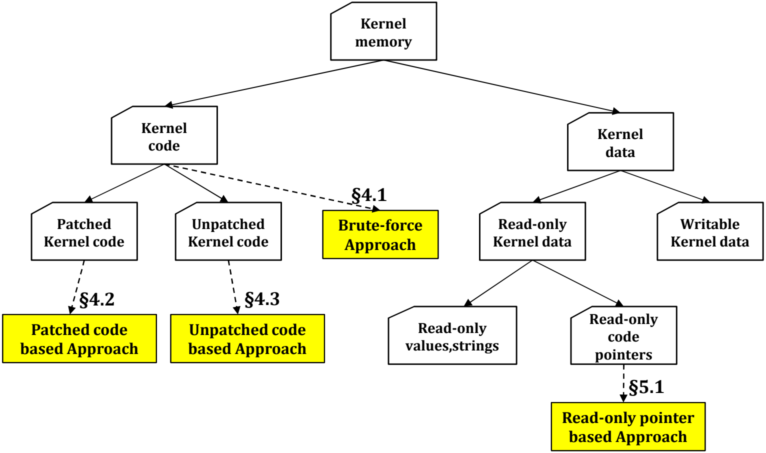
An Overview of the Investigated Approaches



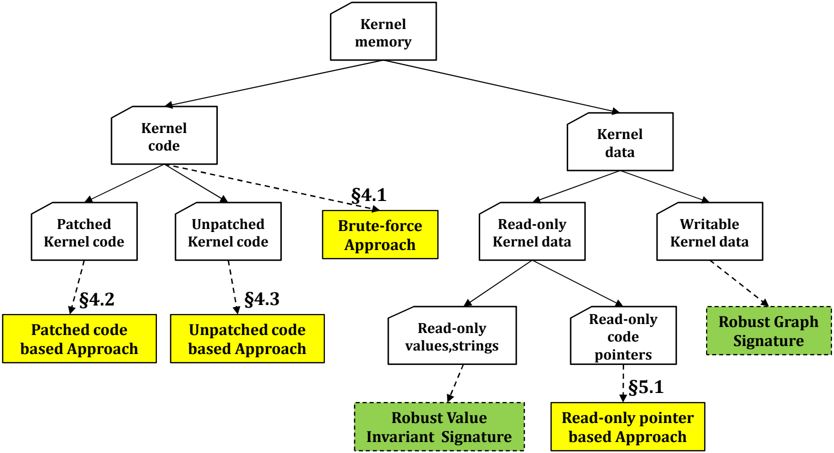
An Overview of the Investigated Approaches



An Overview of the Investigated Approaches



An Overview of the Investigated Approaches



Approach-I: Brute-force Code Matching

Algorithm 1: A Brute-Force Based Code Matching Approach

```
Data: Kernel Page Size 4096;
Input: Kernel memory snapshot: $M$ with $M_p$ pages; Kernel
       code in disk $D$ with $D_p$ pages;
Result: The base address of the randomized kernel code
begin
    $peak \leftarrow 0$;
    $R \leftarrow 0$;
    for $i \in \{0..M_p\}$ do
        $matched \leftarrow 0$;
        $j \leftarrow 0$;
        while $j < D_p$ do
            $M[i] \leftarrow$ GetVirtualPageContent (M, i);
            if ($k \leftarrow$ PageMatching (M[i], D[j]) and
            $k > 2048$) then
                $j \leftarrow j + 1$; $matched \leftarrow matched + k$ ;
            else
                j$\leftarrow$ 0; break;
        if (($j == D_p$) and ($matched/D > peak$)) then
            $peak \leftarrow matched/D$; $R \leftarrow$ GetVirtualAddr
            $(M, i - j)$;
    return $R$;
```

Pros and Cons

- The **simplest** (no sophisticated analysis required), and has very strong robustness since it uses the entire kernel code as the signatures.
- May have **false negatives** when facing strong adversaries.

Approach-II: Using Patched Code (Relocation Entries [ARG15])

```
Code in Disk Image         Base Address: 0xc0100000

c0100450: c7 04 24 d8 15 80 c0      movl    $0xc08015d8,(%esp)
c0100457: 89 44 24 0c               mov     %eax,0xc(%esp)
c010045b: e8 20 31 04 00            call    c0143580
c0100460: e9 3c ff ff ff            jmp     c01003a1
c0100465: 8d 74 26 00               lea     0x0(%esi,%eiz,1),%esi
c0100469: 8d bc 27 00 00 00 00      lea     0x0(%edi,%eiz,1),%edi
c0100470: 55                        push    %ebp
c0100471: 89 e5                     mov     %esp,%ebp
c0100473: e8 08 38 58 00            call    c0683c80
c0100478: a3 80 a7 8c c0            mov     %eax,0xc08ca780
```

```
Code in Memory Snapshot    Base Address: 0xcc200000
cc200450: c7 04 24 d8 15 90 cc      movl    $0xcc9015d8,(%esp)
cc200457: 89 44 24 0c               mov     %eax,0xc(%esp)
cc20045b: e8 20 31 04 00            call    cc243580
cc200460: e9 3c ff ff ff            jmp     cc2003a1
cc200465: 8d 74 26 00               lea     0x0(%esi,%eiz,1),%esi
cc200469: 8d bc 27 00 00 00 00      lea     0x0(%edi,%eiz,1),%edi
cc200470: 55                        push    %ebp
cc200471: 89 e5                     mov     %esp,%ebp
cc200473: 66 66 66 66 90            xchg    %ax,%ax
cc200478: a3 80 a7 9c cc            mov     %eax,0xcc9ca780
```

Approach-II: Using Patched Code (Relocation Entries [ARG15])

```
Code in Disk Image        Base Address: 0xc0100000

c0100450: c7 04 24 d8 15 80 c0      movl   $0xc08015d8,(%esp)
c0100457: 89 44 24 0c               mov    %eax,0xc(%esp)
c010045b: e8 20 31 04 00            call   c0143580
c0100460: e9 3c ff ff ff            jmp    c01003a1
c0100465: 8d 74 26 00               lea    0x0(%esi,%eiz,1),%esi
c0100469: 8d bc 27 00 00 00 00      lea    0x0(%edi,%eiz,1),%edi
c0100470: 55                        push   %ebp
c0100471: 89 e5                     mov    %esp,%ebp
c0100473: e8 08 38 58 00            call   c0683c80
c0100478: a3 80 a7 8c c0            mov    %eax,0xc08ca780
```

```
Code in Memory Snapshot    Base Address: 0xcc200000
cc200450: c7 04 24 d8 15 90 cc      movl   $0xcc9015d8,(%esp)
cc200457: 89 44 24 0c               mov    %eax,0xc(%esp)
cc20045b: e8 20 31 04 00            call   cc243580
cc200460: e9 3c ff ff ff            jmp    cc2003a1
cc200465: 8d 74 26 00               lea    0x0(%esi,%eiz,1),%esi
cc200469: 8d bc 27 00 00 00 00      lea    0x0(%edi,%eiz,1),%edi
cc200470: 55                        push   %ebp
cc200471: 89 e5                     mov    %esp,%ebp
cc200473: 66 66 66 66 90            xchg   %ax,%ax
cc200478: a3 80 a7 9c cc            mov    %eax,0xcc9ca780
```

Approach-II: Using Patched Code (Relocation Entries [ARG15])

```
Code in Disk Image          Base Address: 0xc0100000
c0100450: c7 04 24 d8 15 80 c0      movl   $0xc08015d8,(%esp)
c0100457: 89 44 24 0c               mov    %eax,0xc(%esp)
c010045b: e8 20 31 04 00            call   c0143580
c0100460: e9 3c ff ff ff            jmp    c01003a1
c0100465: 8d 74 26 00               lea    0x0(%esi,%eiz,1),%esi
c0100469: 8d bc 27 00 00 00 00      lea    0x0(%edi,%eiz,1),%edi
c0100470: 55                        push   %ebp
c0100471: 89 e5                     mov    %esp,%ebp
c0100473: e8 08 38 58 00            call   c0683c80
c0100478: a3 80 a7 8c c0            mov    %eax,0xc08ca780
```

```
            Relocation Entries
     Offset     Type        Name
1: c0100453  R_386_32    .rodata
2: c010045c  R_386_PC32  warn_slowpath_fmt
3: c0100474  R_386_PC32  mcount
4: c0100479  R_386_32    .data
```

```
Code in Memory Snapshot   Base Address: 0xcc200000
cc200450: c7 04 24 d8 15 90 cc      movl   $0xcc9015d8,(%esp)
cc200457: 89 44 24 0c               mov    %eax,0xc(%esp)
cc20045b: e8 20 31 04 00            call   cc243580
cc200460: e9 3c ff ff ff            jmp    cc2003a1
cc200465: 8d 74 26 00               lea    0x0(%esi,%eiz,1),%esi
cc200469: 8d bc 27 00 00 00 00      lea    0x0(%edi,%eiz,1),%edi
cc200470: 55                        push   %ebp
cc200471: 89 e5                     mov    %esp,%ebp
cc200473: 66 66 66 66 90            xchg   %ax,%ax
cc200478: a3 80 a7 9c cc            mov    %eax,0xcc9ca780
```

Approach-II: Using Patched Code (Relocation Entries [ARG15])

```
Code in Disk Image          Base Address: 0xc0100000
c0100450: c7 04 24 d8 15 80 c0      movl   $0xc08015d8,(%esp)
c0100457: 89 44 24 0c               mov    %eax,0xc(%esp)
c010045b: e8 20 31 04 00            call   c0143580
c0100460: e9 3c ff ff ff            jmp    c01003a1
c0100465: 8d 74 26 00               lea    0x0(%esi,%eiz,1),%esi
c0100469: 8d bc 27 00 00 00 00      lea    0x0(%edi,%eiz,1),%edi
c0100470: 55                        push   %ebp
c0100471: 89 e5                     mov    %esp,%ebp
c0100473: e8 08 38 58 00            call   c0683c80
c0100478: a3 80 a7 8c c0            mov    %eax,0xc08ca780
```

```
Code in Memory Snapshot     Base Address: 0xcc200000
cc200450: c7 04 24 d8 15 90 cc      movl   $0xcc9015d8,(%esp)
cc200457: 89 44 24 0c               mov    %eax,0xc(%esp)
cc20045b: e8 20 31 04 00            call   cc243580
cc200460: e9 3c ff ff ff            jmp    cc2003a1
cc200465: 8d 74 26 00               lea    0x0(%esi,%eiz,1),%esi
cc200469: 8d bc 27 00 00 00 00      lea    0x0(%edi,%eiz,1),%edi
cc200470: 55                        push   %ebp
cc200471: 89 e5                     mov    %esp,%ebp
cc200473: 66 66 66 66 90            xchg   %ax,%ax
cc200478: a3 80 a7 9c cc            mov    %eax,0xcc9ca780
```

```
        Relocation Entries
   Offset      Type       Name
1: c0100453  R_386_32   .rodata
2: c010045c  R_386_PC32 warn_slowpath_fmt
3: c0100474  R_386_PC32 mcount
4: c0100479  R_386_32   .data
```

CodeIdentifier Approach

$V_d - V_b = S$

$V_m - V_x = S$

```
1: 0xc08015d8 – 0xc0100000 = 0x7015d8
1: 0xcc9015d8 – 0xcc200000 = 0x7015d8
4: 0xc08ca780 – 0xc0100000 = 0x7ca780
4: 0xcc9ca780 – 0xcc200000 = 0x7ca780
```

Approach-II: Using Patched Code (Relocation Entries [ARG15])

Code in Disk Image Base Address: 0xc0100000

```
c0100450: c7 04 24 d8 15 80 c0      movl   $0xc08015d8,(%esp)
c0100457: 89 44 24 0c               mov    %eax,0xc(%esp)
c010045b: e8 20 31 04 00            call   c0143580
c0100460: e9 3c ff ff ff            jmp    c01003a1
c0100465: 8d 74 26 00               lea    0x0(%esi,%eiz,1),%esi
c0100469: 8d bc 27 00 00 00 00      lea    0x0(%edi,%eiz,1),%edi
c0100470: 55                        push   %ebp
c0100471: 89 e5                     mov    %esp,%ebp
c0100473: e8 08 38 58 00            call   c0683c80
c0100478: a3 80 a7 8c c0            mov    %eax,0xc08ca780
```

Code in Memory Snapshot Base Address: 0xcc200000

```
cc200450: c7 04 24 d8 15 90 cc      movl   $0xcc9015d8,(%esp)
cc200457: 89 44 24 0c               mov    %eax,0xc(%esp)
cc20045b: e8 20 31 04 00            call   cc243580
cc200460: e9 3c ff ff ff            jmp    cc2003a1
cc200465: 8d 74 26 00               lea    0x0(%esi,%eiz,1),%esi
cc200469: 8d bc 27 00 00 00 00      lea    0x0(%edi,%eiz,1),%edi
cc200470: 55                        push   %ebp
cc200471: 89 e5                     mov    %esp,%ebp
cc200473: 66 66 66 66 90            xchg   %ax,%ax
cc200478: a3 80 a7 9c cc            mov    %eax,0xcc9ca780
```

Relocation Entries

```
   Offset     Type        Name
1: c0100453   R_386_32    .rodata
2: c010045c   R_386_PC32  warn_slowpath_fmt
3: c0100474   R_386_PC32  mcount
4: c0100479   R_386_32    .data
```

CodeIdentifier Approach

$V_d - V_b = S$

$V_m - V_x = S$

1: 0xc08015d8 – 0xc0100000 = 0x7015d8
1: 0xcc9015d8 – 0xcc200000 = 0x7015d8
4: 0xc08ca780 – 0xc0100000 = 0x7ca780
4: 0xcc9ca780 – 0xcc200000 = 0x7ca780

Our approach

$V_m - V_d = RandomizeOffset$

1: 0xcc9015d8 – 0xc08015d8 = 0x0c100000
4: 0xcc9ca780 – 0xc08ca780 = 0x0c100000

Approach-III: Using Unpatched Patched Code

```
Code in Disk Image           Base Address: 0xc0100000

c0100450: c7 04 24 d8 15 80 c0      movl    $0xc08015d8,(%esp)
c0100457: 89 44 24 0c               mov     %eax,0xc(%esp)
c010045b: e8 20 31 04 00            call    c0143580
c0100460: e9 3c ff ff ff            jmp     c01003a1
c0100465: 8d 74 26 00               lea     0x0(%esi,%eiz,1),%esi
c0100469: 8d bc 27 00 00 00 00      lea     0x0(%edi,%eiz,1),%edi
c0100470: 55                        push    %ebp
c0100471: 89 e5                     mov     %esp,%ebp
c0100473: e8 08 38 58 00            call    c0683c80
c0100478: a3 80 a7 8c c0            mov     %eax,0xc08ca780
```

```
Code in Memory Snapshot      Base Address: 0xcc200000
cc200450: c7 04 24 d8 15 90 cc      movl    $0xcc9015d8,(%esp)
cc200457: 89 44 24 0c               mov     %eax,0xc(%esp)
cc20045b: e8 20 31 04 00            call    cc243580
cc200460: e9 3c ff ff ff            jmp     cc2003a1
cc200465: 8d 74 26 00               lea     0x0(%esi,%eiz,1),%esi
cc200469: 8d bc 27 00 00 00 00      lea     0x0(%edi,%eiz,1),%edi
cc200470: 55                        push    %ebp
cc200471: 89 e5                     mov     %esp,%ebp
cc200473: 66 66 66 66 90            xchg    %ax,%ax
cc200478: a3 80 a7 9c cc            mov     %eax,0xcc9ca780
```

Approach-III: Using Unpatched Patched Code

```
Code in Disk Image          Base Address: 0xc0100000
c0100450: c7 04 24 d8 15 80 c0       movl    $0xc08015d8,(%esp)
c0100457: 89 44 24 0c                mov     %eax,0xc(%esp)
c010045b: e8 20 31 04 00             call    c0143580
c0100460: e9 3c ff ff ff             jmp     c01003a1
c0100465: 8d 74 26 00                lea     0x0(%esi,%eiz,1),%esi
c0100469: 8d bc 27 00 00 00 00       lea     0x0(%edi,%eiz,1),%edi
c0100470: 55                         push    %ebp
c0100471: 89 e5                      mov     %esp,%ebp
c0100473: e8 08 38 58 00             call    c0683c80
c0100478: a3 80 a7 8c c0             mov     %eax,0xc08ca780
```

```
Code in Memory Snapshot      Base Address: 0xcc200000
cc200450: c7 04 24 d8 15 90 cc       movl    $0xcc9015d8,(%esp)
cc200457: 89 44 24 0c                mov     %eax,0xc(%esp)
cc20045b: e8 20 31 04 00             call    cc243580
cc200460: e9 3c ff ff ff             jmp     cc2003a1
cc200465: 8d 74 26 00                lea     0x0(%esi,%eiz,1),%esi
cc200469: 8d bc 27 00 00 00 00       lea     0x0(%edi,%eiz,1),%edi
cc200470: 55                         push    %ebp
cc200471: 89 e5                      mov     %esp,%ebp
cc200473: 66 66 66 66 90             xchg    %ax,%ax
cc200478: a3 80 a7 9c cc             mov     %eax,0xcc9ca780
```

- **Align** two kernel code run-time images, and remove the diffed code
- **Search** code that never patched

Approach-IV: Using Read-Only Pointers

- Pointers in `.rodata` section
- Using `.rel.rodata` to locate them, similary to Approach-II in locating `.rel.text`

Implementation

Approach	Total	Signature Generation	Signature Matching	C++	Python
Brute Force	669	0	32	649	20
Patched Code	807	0	110	759	48
Unpatched Code	817	41	107	756	61
Readonly Pointer	822	0	124	773	49

Table: Implementation Complexity (Units: LOC).

Experiment Setup

- Using 20 Linux kernels from version 3.14 to 4.0.
- Running each of the tested Linux kernels in a VMware Workstation configured with 512M bytes RAM for the guest OS.

Effectiveness: Robustness

OS-kernels	Brute Force		Patched code		Unpatched code		Readonly pointer	
	|Sig|	Bytes/Page	|Sig|	Bytes/Page	|Sig|	Bytes/Page	|Sig|	Bytes/Page
Linux-3.14.8	5,787,280	4,096	278,156	196	225,632	159	331,956	656
Linux-3.14.11	5,788,560	4,096	278,192	196	225,647	159	332,084	656
Linux-3.14.30	5,802,328	4,096	278,900	196	225,933	159	332,416	656
Linux-3.15	5,793,980	4,096	280,476	198	227,514	160	336,204	659
Linux-3.15.2	5,794,108	4,096	280,480	198	227,514	160	336,208	659
Linux-3.15.4	5,794,940	4,096	280,504	198	227,518	160	336,212	659
Linux-3.16	5,844,284	4,096	281,812	197	229,065	160	340,964	658
Linux-3.16.2	5,846,844	4,096	281,840	197	229,084	160	340,956	658
Linux-3.16.6	5,850,044	4,096	281,916	197	229,213	160	341,068	658
Linux-3.17	5,889,452	4,096	284,832	198	230,785	160	344,240	660
Linux-3.17.2	5,889,324	4,096	284,880	198	230,794	160	344,252	660
Linux-3.17.6	5,894,696	4,096	285,416	198	230,886	160	344,396	661
Linux-3.18	5,929,000	4,096	286,508	198	232,155	160	346,384	662
Linux-3.18.2	5,929,704	4,096	286,516	198	232,159	160	346,448	662
Linux-3.18.4	5,930,280	4,096	286,608	198	232,167	160	346,448	662
Linux-3.18.6	5,931,816	4,096	286,612	197	232,242	160	346,480	662
Linux-3.19	5,977,424	4,096	288,156	197	233,339	159	348,064	662
Linux-3.19.2	5,980,280	4,096	288,216	197	233,466	159	348,104	663
Linux-3.19.4	5,982,136	4,096	288,268	197	233,503	159	348,172	663
Linux-4.0	6,015,102	4,096	289,532	197	235,018	160	351,676	656
mean	5,882,580	4,096	283,891	198	230,182	160	342,137	660

Table: Signature Size

Effectiveness: Match Ratio

OS-kernels	Brute Force	Patched code	Unpatched Data	Readonly pointer
Linux-3.14.8	95.45%	100.00%	100.00%	100.00%
Linux-3.14.11	95.45%	100.00%	100.00%	100.00%
Linux-3.14.30	95.46%	100.00%	100.00%	100.00%
Linux-3.15	95.39%	100.00%	100.00%	100.00%
Linux-3.15.2	95.39%	100.00%	100.00%	100.00%
Linux-3.15.4	95.39%	100.00%	100.00%	100.00%
Linux-3.16	95.40%	100.00%	100.00%	100.00%
Linux-3.16.2	95.40%	100.00%	100.00%	100.00%
Linux-3.16.6	95.40%	100.00%	100.00%	100.00%
Linux-3.17	95.39%	100.00%	100.00%	100.00%
Linux-3.17.2	95.39%	100.00%	100.00%	100.00%
Linux-3.17.6	95.39%	100.00%	100.00%	100.00%
Linux-3.18	95.40%	100.00%	100.00%	100.00%
Linux-3.18.2	95.40%	100.00%	100.00%	100.00%
Linux-3.18.4	95.40%	100.00%	100.00%	100.00%
Linux-3.18.6	95.40%	100.00%	100.00%	100.00%
Linux-3.19	95.40%	100.00%	100.00%	100.00%
Linux-3.19.2	95.41%	100.00%	100.00%	100.00%
Linux-3.19.4	95.41%	100.00%	100.00%	100.00%
Linux-4.0	95.41%	100.00%	100.00%	100.00%
mean	95.41%	100.00%	100.00%	100.00%

Performance

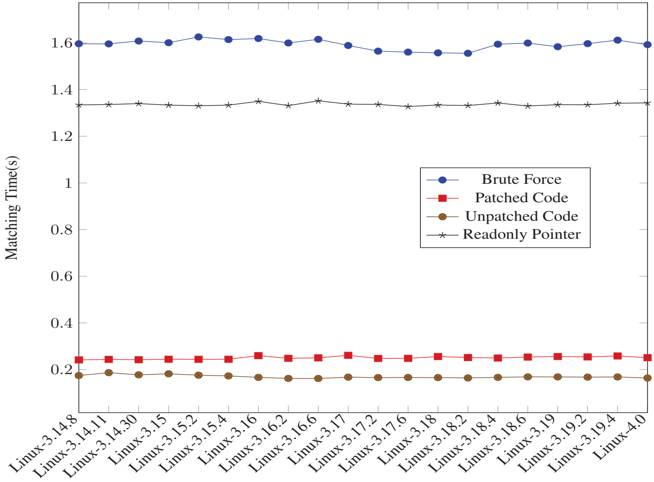


Figure: Signature Matching Performance

Discussions

Open Problems

- When attacker learns the signatures, he/she can generate data with these sigatures though they cannot modify the signatures.
    - e.g., load a copy of the kernel code into the kernel memory
- **Prunning** bogus signatures.

Discussions

Open Problems

- When attacker learns the signatures, he/she can generate data with these sigatures though they cannot modify the signatures.
    - e.g., load a copy of the kernel code into the kernel memory
- **Prunning** bogus signatures.

Future Work

- Writable data (e.g., SigGraph [LRZ+11])
- Other read-only data (e.g., Robust Signatures [DGSTG09])

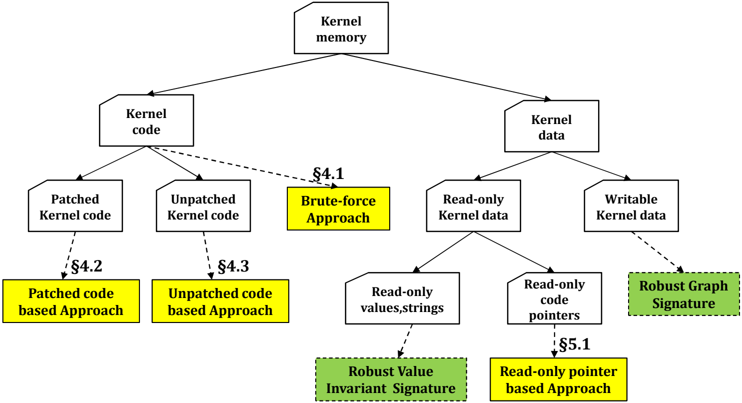
Summary

Derandomizing Kernel ASLR for Introspection and Forensics

We examined the possible optimal approaches from both kernel code and kernel data.

- Brute-force Approach
- Patched code based Approach
- Unpatched code based Approach
- Read-only pointer based Approach

Q&A


Kernel memory

Kernel code

Kernel data

Patched Kernel code

Unpatched Kernel code

§4.1

Brute-force Approach

Read-only Kernel data

Writable Kernel data

§4.2

Patched code based Approach

§4.3

Unpatched code based Approach

Read-only values,strings

Read-only code pointers

Robust Graph Signature

§5.1

Robust Value Invariant Signature

Read-only pointer based Approach

Source code is available at

https://github.com/utds3lab/derandomization

References I

Irfan Ahmed, Vassil Roussev, and Aisha Ali Gombe, Robust fingerprinting for relocatable code, Proceedings of the 5th ACM Conference on Data and Application Security and Privacy, CODASPY 2015, San Antonio, TX, USA, March 2-4, 2015, 2015, pp. 219–229.

Erik Buchanan, Reins Roemer, Hovav Shacham, and Stefan Savage, When good instructions go bad: generalizing return-oriented programming to risc, Proc. 15th ACM Conf. Computer and communications security (CCS'08) (Alexandria, Virginia, USA), ACM, 2008, pp. 27–38.

Brendan Dolan-Gavitt, Abhinav Srivastava, Patrick Traynor, and Jonathon Giffin, Robust signatures for kernel data structures, Proc. 16th ACM Conf. Computer and Communications Security (CCS'09) (Chicago, Illinois, USA), ACM, 2009, pp. 566–577.

Yangchun Fu, Zhiqiang Lin, and David Brumley, Automatically deriving pointer reference expressions from executions for memory dump analysis, Proceedings of the 2015 ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'15) (Bergamo, Italy), September 2015.

Yangchun Fu, Zhiqiang Lin, and Kevin Hamlen, Subverting systems authentication with context-aware, reactive virtual machine introspection, Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC'13) (New Orleans, Louisiana), December 2013.

Tal Garfinkel and Mendel Rosenblum,
A virtual machine introspection based architecture for intrusion detection, Proc. Network and Distributed Systems Security Sym. (NDSS'03), February 2003.

Zhiqiang Lin, Junghwan Rhee, Xiangyu Zhang, Dongyan Xu, and Xuxian Jiang, Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures, Proc. 18th Annual Network and Distributed System Security Sym. (NDSS'11) (San Diego, CA), February 2011.

References II

- Giampaolo Fresi Roglia, Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi, Surgically returning to randomized lib(c), Proceedings of the 25^{th} Annual Computer Security Applications Conference (ACSAC), IEEE Computer Society, December 2009, Honolulu, Hawaii, USA, pp. 60–69.
- Hovav Shacham, The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86), Proc. 14th ACM Conf. Computer and communications security (CCS'07) (Alexandria, Virginia, USA), ACM, 2007, pp. 552–561.
- Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi, Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization, Security and Privacy (SP), 2013 IEEE Symposium on, IEEE, 2013, pp. 574–588.
- Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh, On the effectiveness of address-space randomization, Proceedings of the 11th ACM Conference on Computer and Communications Security (New York, NY, USA), CCS '04, ACM, 2004, pp. 298–307.
- PaX Team, Pax address space layout randomization (aslr), http://pax.grsecurity.net/docs/aslr.txt (2000).
- AAron Walters, The volatility framework: Volatile memory artifact extraction utility framework, 2005, https://www.volatilesystems.com/default/volatility.
- Junyuan Zeng and Zhiqiang Lin, Towards automatic inference of kernel object semantics from binary code, Proceedings of the 18th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'15) (Kyoto, Japan), November 2015.