Hiding in the Shadows:
Empowering ARM for Stealthy Virtual Machine Introspection
ACSAC 2018

Sergej Proskurin, ¹ Tamas Lengyel, ³ Marius Momeu, ¹
Claudia Eckert, ¹ and Apostolis Zarras ²

¹Technical University of Munich
²Maastricht University
³The Honeynet Project

6th of December 2018
Stealthy malware analysis on ARM!
Motivation & Background
Virtual Machine Introspection Recap

Virtual Machine Monitor

VMI-based Monitor
Motivation & Background
Virtual Machine Introspection Recap

Isolation
Inspection
Interposition

VMI-based Monitor

Virtual Machine Monitor

S. Proskurin et al.
ACSAC 2018
Motivation & Background
Virtual Machine Introspection Recap

Isolation

Inspection

Interposition

VMI-based Monitor

Guest VM

Virtual Machine Monitor

ThREAD_INFo
STATE
STACK
USAgE...

TASK_STRUCT

20 00 00 00 00 00 00 00
FF FF FF FF FF FF 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 80 54 0C 00 00 FF FF
02 00 00 00 00 01 40 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
01 00 00 00 01 00 00 00
10 00 00 00 00 00 00 00
BA EC FE FF 00 00 00 00
80 CF 66 28 00 80 FF FF

S. Proskurin et al.
ACSAC 2018
Motivation & Background
Virtual Machine Introspection Recap

Virtual Machine Monitor

VMI-based Monitor

Guest VM

task_struct

thread_info

state

stack

usage

ThREAT_INFo

STATE

STACK

USAGE

...
Motivation & Background
Virtual Machine Introspection Recap

Isolation
Inspection
Interposition

VMI-based Monitor

User-Space:

Kernel-Space:

Virtual Machine Monitor

Isolation
Inspection
Interposition

S. Proskurin et al. ACSAC 2018
Motivation & Background
Virtual Machine Introspection Recap

Isolation
Inspection
Interposition

VMI-based Monitor

User-Space:

Kernel-Space:

```
...]
mov rax, 0x1
syscall
__x64_sys_read:
  int3
  mov rsi, [rdi+0x68]
  mov rdi, [rdi+0x70]
  jmp <ksys_read>
```

Virtual Machine Monitor
Virtual Machine Introspection Recap

Isolation
Inspection
Interposition

VMI-based Monitor

User-Space:

Kernel-Space:

__x64_sys_read:

Guest VM

Virtual Machine Monitor

Motivation & Background

S. Proskurin et al.
ACSAC 2018
Virtual Machine Introspection Recap

**Isolation**

**Inspection**

**Interposition**

**Stealth**

**VMI-based Monitor**

**User-Space:**

```
[...]
mov rax, 0x1
syscall
```

**Kernel-Space:**

```
__x64_sys_read:
  int3
  mov rsi, [rdi+0x68]
  mov rdi, [rdi+0x70]
  jmp <ksys_read>
```

**Guest VM**

**Virtual Machine Monitor**
Virtual Machine Monitor

VMI-based Monitor

Stealth

Isolation

Inspection

Interposition

User-Space:

Kernel-Space:

```
...]
    mov rax, 0x1
    syscall
```

```
_x64_sys_read:
    mov rdx, [rdi+0x60]
    mov rsi, [rdi+0x68]
    mov rdi, [rdi+0x70]
    jmp <ksys_read>
```
The Need for Stealthy Monitoring

Split-personality malware

- Employ anti-virtualization to reveal a VMM (red pills)

Perfect VM transparency is not feasible

- Insufficient to reveal virtual environments alone!

More interesting to know whether the system is being analyzed

→ Hide analysis artifacts from the guest
Requirements for Stealthy Monitoring

1. Intercept the guest in kernel-space
2. A stealthy single-stepping mechanism
3. Execute-only memory
Req. 1: Implementing Kernel Tap Points

Use instructions as a trigger to trap into the VMM
- E.g., software breakpoints (BRK/BKPT instruction)

Better: Secure Monitor Call instruction (SMC)
- Guest is not able to subscribe to SMC traps
- SMC traps do not have to be re-injected into the guest
- Can only be executed in the guest’s kernel

User-Space:

```
[...]
move x8, #0x3f
svc #0x0
```

Kernel-Space:

```
MoV x29, SP
STP x29, x30, [SP, #-64]!
MoV x29, SP
STP x21, x22, [SP, #32]
[...]
```

SyS_read:
Req. 1: Implementing Kernel Tap Points

Use instructions as a trigger to trap into the VMM
- E.g., software breakpoints (BRK/BKPT instruction)

Better: Secure Monitor Call instruction (SMC)
- Guest is not able to subscribe to SMC traps
- SMC traps do not have to be re-injected into the guest
- Can only be executed in the guest’s kernel

User-Space:

```
[...]
mov x8, #0x3f
svc #0x0
```

Kernel-Space:

```
SyS_read:
    smc #0x0
    mov x29, sp
    stp x21, x22, [sp, #32]
[...]
```
Req. 1: Implementing Kernel Tap Points

Use instructions as a trigger to trap into the VMM
- E.g., software breakpoints (BRK/BKPT instruction)

Better: Secure Monitor Call instruction (SMC)
- Guest is not able to subscribe to SMC traps
- SMC traps do not have to be re-injected into the guest
- Can only be executed in the guest’s kernel

Issues: How to remain stealthy and in control?
- Removing tap points introduces race conditions
- No hardware support for stealthy single-stepping

User-Space:

```
[...]  
mov x8, #0x3f
svc #0x0
```

Kernel-Space:

```
SyS_read:  
  smc #0x0
  mov x29, sp
  stp x21, x22, [sp, #32]
  [...]
```
ARM does not support **stealthy** single-stepping

- Attackers can reveal the analysis framework
  - We need a novel, stealthy single-stepping scheme

**Leverage the fixed-width ISA for single-stepping**

- Locate instruction boundaries without a disassembler
- Use two SMCs to single-step one instruction
  - Multi-vCPU safe!
ARM does not support **stealthy** single-stepping

- Attackers can reveal the analysis framework
  - We need a novel, stealthy single-stepping scheme

Leverage the fixed-width ISA for single-stepping

- Locate instruction boundaries without a disassembler
  - Use two SMCs to single-step one instruction
  - Multi-vCPU safe!
ARM does not support **stealthy** single-stepping

- Attackers can reveal the analysis framework
  → We need a novel, stealthy single-stepping scheme

Leverage the fixed-width ISA for single-stepping

- Locate instruction boundaries without a disassembler
- Use two SMCs to single-step one instruction
  → Multi-vCPU safe!
ARM does not support **stealthy** single-stepping

- Attackers can reveal the analysis framework
  - We need a novel, stealthy single-stepping scheme

Leverage the fixed-width ISA for single-stepping

- Locate instruction boundaries without a disassembler
- Use two SMCs to single-step one instruction
  - Multi-vCPU safe!
ARM does not support **stealthy** single-stepping

- Attackers can reveal the analysis framework
- We need a novel, stealthy single-stepping scheme

Leverage the fixed-width ISA for single-stepping

- Locate instruction boundaries without a disassembler
- Use two SMCs to single-step one instruction
- Multi-vCPU safe!
ARM does not support **stealthy** single-stepping

- Attackers can reveal the analysis framework
  - We need a novel, stealthy single-stepping scheme

Leverage the fixed-width ISA for single-stepping

- Locate instruction boundaries without a disassembler
- Use two SMCs to single-step one instruction
  - Multi-vCPU safe!
ARM does not support **stealthy** single-stepping

- Attackers can reveal the analysis framework
  - We need a novel, stealthy single-stepping scheme

Leverage the fixed-width ISA for single-stepping

- Locate instruction boundaries without a disassembler
  - Use two SMCs to single-step one instruction
  - Multi-vCPU safe!

How do we hide injected SMC instructions?

- Employ Second Level Address Translation (SLAT)
Xen physical to machine (p2m) subsystem

- Uses Second Level Address Translation (SLAT)
- Translates guest-physical to machine-physical addresses
- Represents a single view on the guest’s physical memory

Xen p2m allows to control access permissions of the guest’s physical memory

- Hide injected SMC instructions by withdrawing read-permissions
Req. 2: (Stealthy) Single-Stepping

Xen p2m Subsystem

Xen physical to machine (p2m) subsystem
- Uses Second Level Address Translation (SLAT)
- Translates guest-physical to machine-physical addresses
- Represents a single view on the guest’s physical memory

Xen p2m allows to control access permissions of the guest’s physical memory
- Hide injected SMC instructions by withdrawing read-permissions

Issue: On integrity-checks permissions must be relaxed
- Walking the page tables is slow
- Another vCPU can access the memory without notifying the VMM
Xen alternate p2m (altp2m) subsystem

- Maintains different views on the guest’s physical memory
- Allows to allocate and assign different memory views to vCPUs
  → Switch views instead of relaxing permissions in a global view!
Req. 2: (Stealthy) Single-Stepping

Xen alternate p2m (altp2m) Subsystem

- Maintains different views on the guest’s physical memory
- Allows to allocate and assign different memory views to vCPUs
  → Switch views instead of relaxing permissions in a global view!
Xen alternate p2m (altp2m) subsystem

- Allows to remap same guest-physical to different machine-physical page frames
- Facilitates, e.g., SMC injections in selected views
Req. 2: (Stealthy) Single-Stepping

Xen altp2m Subsystem

Issue: No ARM support

を使った

Xen altp2m exclusively used on Intel CPUs
Req. 2: (Stealthy) Single-Stepping
Xen altp2m Subsystem on ARM
Req. 2: (Stealthy) Single-Stepping

Xen altp2m Subsystem on ARM

(a) Without Xen altp2m.
Req. 2: (Stealthy) Single-Stepping

Xen altp2m Subsystem on ARM

(a) Without Xen altp2m.
Req. 2: (Stealthy) Single-Stepping

Xen altp2m Subsystem on ARM

(a) Without Xen altp2m.
Req. 2: (Stealthy) Single-Stepping
Xen altp2m Subsystem on ARM

(a) Without Xen altp2m.
Req. 2: (Stealthy) Single-Stepping
Xen altp2m Subsystem on ARM

(a) Without Xen altp2m.
Req. 2: (Stealthy) Single-Stepping
Xen altp2m Subsystem on ARM

(a) Without Xen altp2m.

(a) With Xen altp2m.
Req. 2: (Stealthy) Single-Stepping
Xen altp2m Subsystem on ARM

Original-View

--x

GFN1 → MFn1
Original Page
SMC
Instr 2
Instr 3
...
VMID
GFN 1
GFN 2
Guest Physical Memory

--x

GFN2 → MFn2
Backup Page
Instr 1
SMC
...
VMID'

(b) Without Xen altp2m.

Execute

--x

GFN ← MFn1
Shadow-Copy'
SMC
Instr 2
Instr 3
...
VMID'

GFN
Guest Physical Memory

Step

--x

GFN ← MFn2
Shadow-Copy''
SMC
Instr 1
Instr 3
...
VMID''

GFN
Guest Physical Memory

(b) With Xen altp2m.
Req. 2: (Stealthy) Single-Stepping
Xen altp2m Subsystem on ARM

(b) Without Xen altp2m.

(b) With Xen altp2m.
Req. 2: (Stealthy) Single-Stepping
Xen altp2m Subsystem on ARM

(b) Without Xen altp2m.

(b) With Xen altp2m.
Req. 2: (Stealthy) Single-Stepping
Xen altp2m Subsystem on ARM

Original-View

�行

(b) Without Xen altp2m.

Execute

(b) With Xen altp2m.
Req. 2: (Stealthy) Single-Stepping
Xen altp2m Subsystem on ARM

Original-View

(b) Without Xen altp2m.

Execute

(b) With Xen altp2m.
Putting everything together (on AArch64)

- Allocate two additional views: **Execute-** and **Step-View**
- Duplicate the original page twice
  - Replace `instr 1` with `SMC in shadow-copy`
  - Replace `instr 2` with `SMC in shadow-copy`"
- Map both duplicates as **execute-only**

On read-requests, switch to the Original-View

- Satisfies integrity checks
1. Intercept the guest in kernel-space
   ✓ Secure Monitor Call (SMC) instruction

2. A stealthy single-stepping mechanism
   ✗ ARM has no hardware support for *stealthy* single-stepping
   ✓ Stealthy single-stepping via Xen altp2m (when combined with execute-only memory)

3. Execute-only memory
   ✓ AArch64
   ✗ AArch32 lacks execute-only memory
   ✓ Splitting the TLBs to hide injected tap points on AArch32
Build the foundation for stealthy monitoring on ARM

- Implement Xen altp2m for ARM
- Equip DRAKVUF and LibVMI with our single-stepping primitives

Use DRAKVUF to trace system calls inside the guest VM

- HiKey LeMaker development board
- Guest runs a Linux v4.15 kernel
- Xen v4.11
**Table:** Monitoring overhead (OHD) of DRAKVUF utilizing Hardware-SS, Double-SMC-SS, and Split-TLB-SS primitives measured by Lmbench 3.0, in msec.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>w/o</th>
<th>Hardware</th>
<th>(OHD)</th>
<th>Step-View</th>
<th>Double-SMC</th>
<th>Split-TLB</th>
<th>Backup Page</th>
<th>(OHD)</th>
<th>Backup Page</th>
<th>(OHD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fork+execve</td>
<td>1383.33</td>
<td>6053.67</td>
<td>4.38 x</td>
<td>5567.33</td>
<td>4.02 x</td>
<td>6030.00</td>
<td>4.36 x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fork+exit</td>
<td>377.43</td>
<td>835.52</td>
<td>2.21 x</td>
<td>787.14</td>
<td>2.09 x</td>
<td>924.83</td>
<td>2.45 x</td>
<td>5910.83</td>
<td>15.66 x</td>
<td>4225.83</td>
</tr>
<tr>
<td>fork+/bin/sh</td>
<td>3249.17</td>
<td>12542.00</td>
<td>3.86 x</td>
<td>11672.67</td>
<td>3.59 x</td>
<td>12737.33</td>
<td>3.92 x</td>
<td>53134.66</td>
<td>16.35 x</td>
<td>34231.33</td>
</tr>
<tr>
<td>fsck</td>
<td>0.62</td>
<td>94.94</td>
<td>152.57 x</td>
<td>78.65</td>
<td>126.40 x</td>
<td>84.20</td>
<td>135.81 x</td>
<td>103.52</td>
<td>166.97 x</td>
<td>75.33</td>
</tr>
<tr>
<td>mem read</td>
<td>1745.00</td>
<td>1692.33</td>
<td>0.97 x</td>
<td>1692.33</td>
<td>0.97 x</td>
<td>1738.00</td>
<td>1.00 x</td>
<td>1730.33</td>
<td>0.99 x</td>
<td>1735.33</td>
</tr>
<tr>
<td>mem write</td>
<td>4687.67</td>
<td>4308.33</td>
<td>0.92 x</td>
<td>4308.33</td>
<td>0.92 x</td>
<td>4715.00</td>
<td>1.00 x</td>
<td>4575.33</td>
<td>0.98 x</td>
<td>4602.00</td>
</tr>
<tr>
<td>open/close</td>
<td>5.44</td>
<td>202.67</td>
<td>37.25 x</td>
<td>158.33</td>
<td>29.11 x</td>
<td>179.26</td>
<td>35.95 x</td>
<td>269.67</td>
<td>49.57 x</td>
<td>184.65</td>
</tr>
<tr>
<td>page fault</td>
<td>1.49</td>
<td>1.72</td>
<td>1.15 x</td>
<td>1.74</td>
<td>1.16 x</td>
<td>1.62</td>
<td>1.09 x</td>
<td>1.90</td>
<td>1.28 x</td>
<td>1.91</td>
</tr>
<tr>
<td>pipe lat</td>
<td>12.26</td>
<td>371.92</td>
<td>30.34 x</td>
<td>344.83</td>
<td>28.13 x</td>
<td>425.28</td>
<td>34.69 x</td>
<td>955.53</td>
<td>77.94 x</td>
<td>482.60</td>
</tr>
<tr>
<td>read</td>
<td>0.67</td>
<td>95.21</td>
<td>141.14 x</td>
<td>79.10</td>
<td>117.27 x</td>
<td>84.06</td>
<td>125.46 x</td>
<td>99.34</td>
<td>148.27 x</td>
<td>75.39</td>
</tr>
<tr>
<td>select 500 fd</td>
<td>28.33</td>
<td>124.62</td>
<td>4.40 x</td>
<td>110.23</td>
<td>3.89 x</td>
<td>114.51</td>
<td>4.04 x</td>
<td>124.47</td>
<td>4.39 x</td>
<td>113.85</td>
</tr>
<tr>
<td>signal handle</td>
<td>4.34</td>
<td>189.67</td>
<td>43.70 x</td>
<td>150.33</td>
<td>34.64 x</td>
<td>154.13</td>
<td>35.51 x</td>
<td>178.00</td>
<td>41.01 x</td>
<td>158.33</td>
</tr>
<tr>
<td>signal install</td>
<td>0.51</td>
<td>95.00</td>
<td>186.27 x</td>
<td>72.00</td>
<td>141.18 x</td>
<td>75.13</td>
<td>147.31 x</td>
<td>89.07</td>
<td>174.65 x</td>
<td>73.73</td>
</tr>
<tr>
<td>stat</td>
<td>2.63</td>
<td>99.97</td>
<td>38.06 x</td>
<td>80.73</td>
<td>30.74 x</td>
<td>85.30</td>
<td>32.43 x</td>
<td>105.58</td>
<td>40.14 x</td>
<td>83.57</td>
</tr>
<tr>
<td>syscall</td>
<td>0.31</td>
<td>94.21</td>
<td>299.05 x</td>
<td>75.15</td>
<td>238.55 x</td>
<td>83.49</td>
<td>269.32 x</td>
<td>98.48</td>
<td>317.68 x</td>
<td>78.84</td>
</tr>
<tr>
<td>write</td>
<td>0.47</td>
<td>95.34</td>
<td>203.32 x</td>
<td>76.82</td>
<td>163.81 x</td>
<td>83.86</td>
<td>178.43 x</td>
<td>103.22</td>
<td>219.62 x</td>
<td>73.77</td>
</tr>
</tbody>
</table>
Establish the foundation for stealthy malware analysis on ARM

- Introduce Xen altp2m to ARM
- Stealthy single-stepping approach for AArch{32|64}
- De-synchronize the TLB architecture on AArch32

DRAKVUF on ARM is open-source:

- https://github.com/dRAKVUF-on-ARM/dRAKVUF-on-ARM
- https://youtu.be/mfhZBBdC-Jg (Demo!)
Appendix 0: (Stealthy) Single-Stepping

Single-Stepping on ARM

ARM does not support **stealthy** single-stepping

→ Attackers can infer the presence of the analysis framework

**AArch32**: Use **hardware breakpoints** ("mismatching") for single-stepping

- CPU generates a debug event on instructions following the breakpoint
- Finite number of hardware breakpoints

**AArch64**: Use **Software-Step** exceptions (set MDSCR_EL1.SS and PSTATE.SS of EL1)

- ARM forbids access to PSTATE.SS in all exception levels
- Spill PSTATE.SS into the guest-accessible SPSR_EL1
Appendix 1: Xen altp2m Subsytem on Intel

Xen altp2m exclusively used on Intel

- The VMCS has capacity for up to 512 EPTPs (memory views)
- Introduced to Xen to add support for the EPTP Switching functionality
  - Combine VMFUNC instruction with Virtualization Exceptions #VE
  - No additional VM Exit overhead on memory violations!

External monitors can use altp2m

- Unique tool for VMI applications
AArch32 does not support execute-only memory
  ▸ Code-pages must be executable and readable

ARM uses VMIDs as TLB-tags to isolate translations
  ▸ Allocate two views with same VMID to de-synchronize the iTLB from the dTLB

Prime the TLBs in Original-View:
  ▸ iTLB holds the SMC from Execute-View
  ▸ dTLB holds instr 1 from Original-View
Appendix 2: No Execute-only Memory on AArch32

Splitting the TLBs

AArch32 does not support execute-only memory
  ▪ Code-pages must be executable and readable

ARM uses VMIDs as TLB-tags to isolate translations
  ▪ Allocate two views with same VMID to de-synchronize the iTLB from the dTLB

Prime the TLBs in Original-View:
  ▪ iTLB holds the SMC from Execute-View
  ▪ dTLB holds instr 1 from Original-View