Myth and Truth about Hypervisor-Based Kernel Protector:
The Reason Why You Need Shadow-Box

Seunghun Han, Jungwhan Kang
(hanseunghun || ultract)@nsr.re.kr
Who are we?

- Senior security researcher at NSR (National Security Research Institute of South Korea)
- Speaker at HITBSecConf 2016
- Author of the book series titled “64-bit multi-core OS principles and structure, Vol.1&2”
- a.k.a kkamagui, @kkamagui1

- Researcher at NSR
- Participated final round of some CTFs (Codegate, ISEC… held in South Korea)
- Interested in OS security and reading write-up of CTFs
- Got married last year 😊
- a.k.a ultract, @ultracttt
Goal of This Presentation

- We present lightweight hypervisor-based kernel protector, “Shadow-box”
  - Shadow-box defends kernel from security threats efficiently, and we made it from scratch

- We share lessons learned from deploying and operating Shadow-box in real world systems
  - We have been operating Shadow-box since last year and share lessons learned!
Background

Design

Implementation

Lessons Learned
(and the Truth Previous Researches Did Not Tell You)

Demo. and Conclusion
(Black Hat Sound Bytes)
Linux Kernel Is Everywhere!
Security Threats of Linux Kernel

- The Linux kernel suffers from rootkits and security vulnerabilities
  - Rootkits: EnyeLKM, Adore-ng, Sebek, suckit, kbeast, and so many descendants

Devices which use Linux kernel share security threats
Melee Combats at the Kernel-level

- Kernel-level (Ring 0) protections are not enough
  - Lots of rootkits and exploits work in the Ring 0 level
  - Protections against them are often easily bypassed and neutralized
    - Kernel Object Hooking (KOH)
    - Direct Kernel Object Manipulation (DKOM)

Protections need an even lower level (Ring -1)
# Well-known Rootkits

<table>
<thead>
<tr>
<th>Name</th>
<th>Modified Kernel Object</th>
<th>Type</th>
<th>Attribute</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>EnyeLKM 1.3</td>
<td>syscall_trace_entry&lt;br&gt;sysenter_entry&lt;br&gt;module-&gt;list&lt;br&gt;init_net-&gt;proc_net-&gt;subdir-&gt;tcp_data-&gt;tcp4_seq_show</td>
<td>Code&lt;br&gt;Code&lt;br&gt;Data&lt;br&gt;Function pointer</td>
<td>Static&lt;br&gt;Static&lt;br&gt;Dynamic&lt;br&gt;Dynamic</td>
<td>code change, syscall hook, direct kernel object manipulation (DKOM)</td>
</tr>
<tr>
<td>Adore-ng 0.56</td>
<td>vfs_root-&gt;f_op-&gt;write&lt;br&gt;vfs_root-&gt;f_op-&gt;readdir&lt;br&gt;vfs_proc-&gt;f_dentry-&gt;d_inode-&gt;i_op-&gt;lookup&lt;br&gt;socket_udp-&gt;ops-&gt;recvmsg</td>
<td>Function pointer&lt;br&gt;Function pointer&lt;br&gt;Function pointer</td>
<td>Dynamic&lt;br&gt;Dynamic&lt;br&gt;Dynamic</td>
<td>function pointer hook</td>
</tr>
<tr>
<td>Sebek 2.0</td>
<td>sys_call_table&lt;br&gt;vfs_proc_net_dev-&gt;get_info&lt;br&gt;vfs_proc_net_packet-&gt;proc_fops&lt;br&gt;module-&gt;list</td>
<td>System table&lt;br&gt;Function pointer&lt;br&gt;Data</td>
<td>Static&lt;br&gt;Dynamic&lt;br&gt;Dynamic</td>
<td>syscall hook, function pointer hook, DKOM</td>
</tr>
<tr>
<td>Suckit 2.0</td>
<td>idt_table&lt;br&gt;sys_call_table</td>
<td>System table&lt;br&gt;System table</td>
<td>Static&lt;br&gt;Static</td>
<td>idt hook, syscall hook</td>
</tr>
<tr>
<td>kbeast v1</td>
<td>sys_call_table&lt;br&gt;init_net-&gt;proc_net-&gt;subdir-&gt;tcp_data-&gt;tcp4_seq_show&lt;br&gt;module-&gt;list</td>
<td>System table&lt;br&gt;Function pointer&lt;br&gt;Data</td>
<td>Static&lt;br&gt;Dynamic&lt;br&gt;Dynamic</td>
<td>syscall hook, function pointer hook, DKOM</td>
</tr>
</tbody>
</table>

Other rootkits also have similar patterns
Taking the Higher Ground

- Leveraging virtualization technology (VT)
  - VT separates a machine into a host (secure world) and a guest (normal world)
  - The host in Ring -1 can freely access/control the guest in Ring 0 (the converse doesn’t hold)
  - VT-equipped HW: Intel VT-x, AMD AMD-v, ARM TrustZone
Trends of Introducing Ring -1

Host (Secure World)  Virtualization Technology (T.Z., VT-x, AMD-v)  Guest (Normal World)

User  Monitor, control  User

Kernel  Host OS  Guest OS

Host OS  Guest OS
Previous Researches...

**SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes**

Arvind Seshadri  
CyLab/CMU  
Pittsburgh, PA, USA  
arvinda@cs.cmu.edu

Mark Luk  
CyLab/CMU  
Pittsburgh, PA, USA  
mluk@ece.cmu.edu

Ning Qu  
CyLab/CMU  
Pittsburgh, PA, USA  
quing@cmu.edu

Adrian Perrig  
CyLab/CMU  
Pittsburgh, PA, USA  
perrig@cmu.edu

**Abstract**

We propose SecVisor, a tiny hypervisor that ensures code integrity for commodity OS kernels. In particular, SecVisor enforces that only user-approved code can execute in kernel mode over the entire system lifetime. This protects the kernel against code injection attacks, such as kernel rootkits. SecVisor can achieve this property even against an attacker who controls everything but the CPU, the memory controller, and system memory chips. Further, SecVisor can even defend against attackers with knowledge of zero-day kernel exploits.

Our goal is to make SecVisor immune to formal verification.

**1. INTRODUCTION**

Computing platforms are steadily increasing in complexity, incorporating an ever-growing range of hardware and supporting an ever-growing range of applications. Consequently, the complexity of OS kernels is steadily increasing. The increased complexity of OS kernels also increases the number of security vulnerabilities. The effect of these vulnerabilities is compounded by the fact that, despite many efforts to make kernels modular, most kernels in common use today are monolithic in their design. A compromise of any part of a monolithic kernel could compromise the entire kernel. Since the kernel occupies a privileged position in the software stack...
I heard and knew about them
But, I can not find in real world!
Restrictions on Previous Researches (1)

- Many researches have **preconditions**
  - They usually change kernel code or hypervisor
  - They also need well-known hashes of LKM, well-known value of kernel data, secure VM for analyzing target VM, etc.

- Many researches consume **much resource**
  - The host and the guest run each OS
    - They allocate resources independently!
  - The host consumes many CPU cycles to introspect the guest because of semantic gap
Restrictions on Previous Researches (2)

- In conclusion, previous researches are considered for laboratory environment only
  - They assume they can control environment!
  - But, real world environment is totally different from laboratory environment!
  - You even don’t know the actual environment before the software is installed!
Therefore, a PRACTICAL and LIGHTWEIGHT mechanism is needed for the REAL WORLD ENVIRONMENT!
Design Goals of Kernel Protector

- **Lightweight**
  - Focus on rootkit detection and protection
    - Simple and extensible architecture
  - Small memory footprint
    - No secure VMs and no multiple OSes

- **Practical**
  - Out-of-box approach
    - No modification of kernel code and data
  - Dynamic injection
    - Load any time from boot to runtime
Background
Design
Implementation
Lessons Learned
(and the Truth Previous Researches Did Not Tell You)
Demo. and Conclusion
(Black Hat Sound Bytes)
Security Architecture in Shadow Play
We named this architecture “Shadow-box”
Architecture of Shadow-Box

Host (Ring -1)
- Shadow-Watcher (Monitor)
- Shared Kernel (Read/Write Permission)
- Shared Kernel Only

Shared Area

Monitor, control

Guest (Ring 0~3)
- User (Read/Write Permission)
- Shared Kernel (Read-only Permission)
- Shared Kernel and User

Light-Box (Lightweight Hypervisor)
Architecture of Light-Box

- Light-box, lightweight hypervisor,
- Isolates worlds by using memory protection technique in VT
- Shares the kernel area between the host (Ring -1) and the guest (Ring 0 ~ 3)
  - Does not run each OS in two worlds
- Uses smaller resources than existing mechanisms and has narrow semantic gap
- Can be loaded any time (loadable kernel module)
Architecture of Shadow-Watcher

- Shadow-watcher
  - Monitors the guest by using Light-box
  - Checks if applications of the guest modify kernel objects or not by event-driven way
    - Code, system table, IDT table, etc.
  - Checks the integrity of the guest by introspecting kernel object by periodic way
    - Process list, loadable kernel module (LKM) list, function pointers of file system and socket
What can Shadow-Box do?

- **Shadow-box protects Linux kernel from**
  - **Static kernel object attacks**
    - Static kernel object = immutable in runtime
    - Code modification and system table modification attacks
  - **Dynamic kernel object attacks**
    - Dynamic kernel object = mutable in runtime
    - Process hiding and module hiding
    - Function pointer modification attacks
Background

Design

Implementation

Lessons Learned
(and the Truth Previous Researches Did Not Tell You)

Demo. and Conclusion
(Black Hat Sound Bytes)
Boot Process using Shadow-Box

1. Starting UEFI with Secure Boot
2. Starting Bootloader
3. Starting Linux Kernel
4. Loading Shadow-Box
5. Preparing Virtualization
   - Enabling VMX (Virtual Machine Extension)
   - Identifying kernel information
   - Setting VMCS (Virtual Machine Control Structure)
6. Separating and Starting the Guest
   - Separating memory area
   - Launching VMCS
7. Guest (Normal World)
8. Starting Linux Applications
9. Host (Secure World)
10. Monitoring the Guest

Legend:
- : Linux
- : Shadow-Box
Static Kernel Object Protection (1)

CPU

Guest Physical Address

VT-x Extended Page Table (EPT)

Level 4
  Level 3
    Level 2
      Level 1

Paging Structure of EPT

Host Physical Address

User Area

Static Kernel Objects

Shadow Box Objects

Locking (Readable & Executable)

Hiding (Inaccessible)

Read, Write, Execute

Read, Execute

No Permission

Physical

Physical

Physical

Physical

Physical
Static Kernel Object Protection (2)

Guest (Normal World) Address Translation (Ring 0)

Guest Logical Address (GLA)

Page 1
Page 2
Page 3

Guest Page Table (GPT)

Read, Execute
Read, Execute
Read, Execute

Guest Physical Address (GPA)

Page 1
Page 2
Page 3

Host (Secure World) Address Translation (Ring -1)

Extended Page Table (EPT)

Read, Execute
No Permission
Read, Execute

Host Physical Address (HPA)

Page 1
Page 2
Page 3

Guest (Normal World) Address Translation (Ring 0)

Guest Logical Address (GLA)

Page 1
Page 2
Page 3

Guest Page Table (GPT)

Read, Execute
Read, Execute
Read, Execute

Guest Physical Address (GPA)

Page 1
Page 2
Page 3

Host (Secure World) Address Translation (Ring -1)

Extended Page Table (EPT)

Read, Execute
No Permission
Read, Execute

Host Physical Address (HPA)

Page 1
Page 2
Page 3
Static Kernel Object Protection (3)

Guest (Normal World) Address Translation (Ring 0)

Guest Page Table (GPT)
- Guest Logical Address (GLA)
  - Page 1
  - Page 2
  - Page 3

Read, Execute
Read, Execute
Write, Execute

Host (Secure World) Address Translation (Ring -1)

Extended Page Table (EPT)
- Host Physical Address (HPA)
  - Page 1
  - Page 2
  - Page 3

Read, Execute
No Permission
Read, Execute

EPT protects the host from attack propagation of the guest
Static Kernel Object Protection (4)

Paging Structure of Second Level Page Table

VT-d DMA Remapping Reporting (DMAR) Table

DMA

Root Table

Context Table

Level 4

Level 3

Level 2

Level 1

Physical

Physical

Physical

No Permission

Read, Write

Physical Address

User Area

Static Kernel Objects

Shadow Box Objects

No Permission

Hiding (Inaccessible)

Hiding (Inaccessible)
Dynamic Kernel Object Protection (1)

Task and Module List in Guest

① Creating initial data

② Inserting H/W breakpoint

③ Monitoring

④ Shadowing list data

⑤ Comparing data

Next
Prev
Next
Prev
Next
Prev

Task and Module List in Shadow-box

Task and Module Create Function

- do_fork() or load_module()
  - create_object();
  - modify_list();

Task and Module Delete Function

- release_task() or delete_module()
  - delete_object();
  - modify_list();
Dynamic Kernel Object Protection (2)

VFS and Socket Objects of Guest

FP Pointer

Function Pointer Structure

Open
Read
Write
Close
...

Host Logical Address

Invalid

Malicious Code

Invalid

Kernel Code

Valid

Module Code

Valid

Malicious Code
(Loaded after Shadow-box)

Invalid

Kernel Area

User Area

: Code area loaded before Shadow-box
Privileged Register Protection

- GDTR, LDTR and IDTR change interactions between kernel and user level
- IA32_SYSENTER_CS, IA32_SYSENTER_ESP, IA32_STAR, IA32-LSTAR and IA32_FMASK MSR also change them
- These privileged registers are rarely changed after boot!

- So, Shadow-box
  - Locks the privileged registers
  - Locks and Monitors GDT, LDT, and IDT table
## Rootkit Detection

- All rootkits are detected

<table>
<thead>
<tr>
<th>Name</th>
<th>Detected?</th>
<th>Detected Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>EnyeLKM</td>
<td>√</td>
<td>code change, module hide</td>
</tr>
<tr>
<td>Adore-ng 0.56</td>
<td>√</td>
<td>function pointer change, module hide</td>
</tr>
<tr>
<td>Sebek 2.0</td>
<td>√</td>
<td>system table change, module hide</td>
</tr>
<tr>
<td>Suckit 2.0</td>
<td>√</td>
<td>system table change</td>
</tr>
<tr>
<td>kbeast</td>
<td>√</td>
<td>system table change, module hide</td>
</tr>
</tbody>
</table>
Performance Measurements of Prototype

- Application benchmarks show 1% ~ 10% performance overhead
  - 5.3% at kernel compile in single-core processor
  - 6.2% at kernel compile in multi-core processor

Results of Application Benchmark. Lower is better.
(Intel i7-4790 4core 8thread 3.6GHz, 32GB RAM, 512GB SSD)
Lessons Learned
(and the Truth Previous Researches Did Not Tell You)
Ready to launch!

We deployed Shadow-box in REAL WORLD! and ...
We met **BEASTS of REAL WORLD!**

(false positive, slow-down, system hang, etc.)

Previous researches **did not** tell us something important!

**NICE TO MEET YOU**

AGAIN!

**OH, NO...**

**WHAT HAPPENED...**
Lessons Learned - 1

- Code is not immutable!
  - Linux kernel has a `CONFIG_JUMP_LABEL` option!
  - If this option is set, Linux kernel patches itself on runtime!
  - Unfortunately, this option is set by default!

- Solution
  - Option 1: Add exceptional cases for mutable code pages
  - Option 2: If you can build kernel, Turn Off `CONFIG_JUMP_LABEL` option NOW!
Lessons Learned - 2

- Cache type in EPT is very important!
  - Linux system has some memory mapped I/O area
    - BIOS area, APIC area, PCI area, etc.
  - Misconfiguration makes various problems such as system hang, slow down, video mode change error, etc.

- Solution
  - Set uncachable type by default
  - Set write-back type to “System RAM” area only!
Lessons Learned - 2

```
user$ cat /proc/iomem
00000000-00000fff : reserved
00001000-0009dbff : System RAM
0009dc00-0009fff : reserved
000a0000-000bffff : PCI Bus 0000:00
000c0000-000cf7ff : Video ROM
 000c4000-000cbfff : PCI Bus 0000:00
000ce800-000cefff : Adapter ROM
000cf000-000cf7ff : Adapter ROM
000cf800-000d53ff : Adapter ROM
000d5800-000d67ff : Adapter ROM
000e0000-000ffffff : reserved
 000f0000-000ffffff : System ROM
00100000-ca336fff : System RAM
 01000000-01519400 : Kernel code
 01519401-018edfff : Kernel data
 01a21000-01af2fff : Kernel bss
ca337000-cb68bfff : reserved
cb68c000-cbefefff : ACPI Non-volatile Storage
cbeff000-cbfcefff : ACPI Tables
  cbfc000-cbf8ffff : System RAM
 d0000000-dfffffff : PCI MMC/CFG 0000 [bus 00-ff]
 d0000000-dfffffff : reserved
e0000000-f7ffbff : PCI Bus 0000:00
e0000000-f1ffffff : PCI Bus 0000:04
e0000000-effffff : 0000:04:00.0
f0000000-fffffff : 0000:04:00.0
```
Lessons Learned - 3

- **Multi-core environment** is more complicated than you think!
  - Each core modifies process list and module list concurrently
    - When H/W breakpoint exception occurred, other cores could be changing the lists already!
  - So, we need a mechanism for synchronizing lists

- **Solution**
  - Lock `tasklist_lock` and `module_mutext` of the guest while Shadow-box is checking the lists!
Now,
We have been operating
Shadow-box in REAL WORLD SUCCESSFULLY!
Background
Design
Implementation
Lessons Learned
(and the Truth Previous Researches Did Not Tell You)

Demo. and Conclusion
(Black Hat Sound Bytes)
DEMO

SHADOW-BOX

Lightweight Hypervisor-Based Kernel Protector

Shadow-box: CPU Count 4
Shadow-box: Booting CPU ID 0
Shadow-box: Protect Kernel Code Area [*] Complete
Shadow-box: Protect Module Code Area [*] Complete
Shadow-box: Framework Preinitialize [*] Complete
Shadow-box: Framework Initialize [*] Task count 216
Shadow-box: [*] Module count 88 [*] Complete
Shadow-box: Lock IOMMU [*] Lock IOMMU complete
Shadow-box: Execution Complete
Shadow-box: ErrorCode: 0
Future Work

**Linux**

**Shadow-Box**

**VT-x, VT-d**
*Virtualization Technology*

**TrustZone**
*Virtualization Technology*

**intel**

**ARM**

**Multi-platform Support!**
Conclusion and Black Hat Sound Bytes

- Kernel-level (Ring 0) threats should be protected in a more privileged level (Ring -1)
  - We create Ring -1 level by using VT from scratch

- Shadow-box is lightweight and practical
  - Shadow-box uses less resource than existing mechanisms and protects kernel from rootkits

- Real world is Serengeti!
  - Real world is different from laboratory environment
  - You should have a strong mentality for defeating beasts of real world! or use Shadow-box instead!
THE CHOICE IS YOURS!

☐ TRUTH
☐ MYTH

THANK YOU

Project: github.com/kkamagui/shadow-box-for-x86
Contact: hanseunghun@nsr.re.kr, @kkamagui1
        ultract@nsr.re.kr, @ultracctt