Monitoring Hypervisor Integrity at Runtime

Student: Cuong Pham
PIs: Prof. Zbigniew Kalbarczyk, Prof. Ravi K. Iyer
ACC Meeting, Oct 2015
Motivation - Server Virtualization Trend

x86 servers were virtualized in 2012

Source: 451 Research’s TheInfoPro service reports

x86 servers were virtualized in 2012

Building Secure & Reliable VMs

- Chain of trust must be built from **bottom up**
- … and **continuously through time**
Building Secure & Reliable VMs

- Chain of trust must be built from **bottom up**
- … and **continuously through time**
Protect Hypervisors: Existing approaches

- Did I say “continuously through time”?

Diagram:
- **Layer**:
  - VM
  - OS/Hypervisor
  - Firmware/Bios
  - Hardware
  - Intel TXT
  - HyperSentry/SICE ...
  - TPM
  - Physical security

- **Time**:
  - Load-time
  - Execution
  - Periodically measure hypervisor
  - Vulnerable to transient attacks

- **Attack Surface**

5
Introducing hShield

- Assumption: Hardware is trusted
  - TPM, Intel TXT are enabled
  - Physical security

Layer

- VM
- VM
- VM

- OS/Hypervisor
- Firmware/Bios
- Hardware

- Intel TXT
- TPM

- hShield
- Physical security

- Attack Surface

- HProbes
- HyperTap

Continuously measure hypervisor integrity
Threat Model: VM Escape Attacks

- Hardware Assisted Virtualization
- Attackers have full control of guest OS
- Violate hypervisor Control Flow Integrity (CFI)
Example: Venom VM Attack (CVE-2015-3456)

Description
Though the VENOM vulnerability is also agnostic of the guest operating system, an attacker (or an attacker’s malware) would need to have administrative or root privileges in the guest operating system in order to exploit VENOM

**Impact Type:** Allows unauthorized disclosure of information; Allows unauthorized modification; Allows disruption of service

Threat Model: VM Escape Attacks

- **Attack code**
  - DoS host
  - Access other co-located VMs (e.g., sniffing network traffic, stealing images)
  - Install backdoors, access secrets in host…
VM Escape-enabling CVEs...

- **CVE-2007-1744** – Directory traversal vulnerability in shared folders feature
- **CVE-2008-0923** – Path traversal vulnerability in VMware’s shared folders implementation
- **CVE-2009-1244** – Cloudburst (VMware virtual video adapter vulnerability)
- **CVE-2012-0217** – 64-bit PV guest privilege escalation vulnerability
- **CVE-2014-0983** – Oracle VirtualBox 3D acceleration multiple memory corruption vulnerabilities
- **CVE-2015-(2336-2340)** – Escaping VMware Workstation through COM1 (5 CVE!!!)
- **CVE-2015-3456** – QEMU heap overflow flaw in floppy disk driver
- **CVE-2015-5154** – QEMU heap overflow flaw while processing certain ATAPI commands.
hShield Design Goals

1. Resistance to zero-day VM escape attacks
2. Detect both transient and persistent attacks
3. Small performance overhead in attack-free executions
4. Support target randomization
hShield Approach

- **Continuous monitoring**
  - Detect both persistent and transient attacks

- **White-list monitoring**
  - Detect unknown attacks

- **Hardware extension**
  - Hardware isolation and performance
hShield Overview

Detect a VM-escape attack right at the end of the exploited VM-exit: defeat transient attacks.
White-list vs. Black-list

- **White-list**
  - *No one* can access except the white-listed
  - Prevents unknown attacks
  - E.g., Control Flow Integrity (CFI) techniques

- **Black-list**
  - *Everyone* can access except the black-listed
  - Prevents known (black-listed) attacks
  - E.g., Signature-based malware detection

![Control Flow Graph (CFG)](image)
Current CFI Techniques

• **Heavily relies on static analysis to construct CFG**
  - No interactions with dynamic libraries, OS
  - Heuristic (e.g., pointer analysis is imperfect)
  - Scalability issues (e.g., large binaries)

• **High runtime overhead**
  - Check at every branch

• **Compatibility issues against**
  - Address Space Layout Randomization (ASLR)
  - Programs relying on dynamic binary re-writing (e.g., Linux kernels)
hShield Approach

- **Dynamic analysis to construct CFG**
  - Full system coverage
  - Load-time binary rewriting compatible
- **Check at the sink of executions**
  - Reduce runtime overhead
- **Use hashes of basic blocks instead of addresses**
  - ASLR compatible
hShield Approach

Workloads
VM
Hypervisor

Most popular paths

HotTable
$H_1$
$H_2$

... 
$H_t$

Hash $H_i$

Find $H_i$ in HotTable
Yes

Recomputed $H_i$ in CFG
Yes

Bad

Good

Profiling

Runtime checking

CFG Construction

Profiling

Runtime checking

VM-exit

1

2

3

5

6

VM-entry

1

2

3
Profiling Result: Path Popularity

Setup: Qemu (HW) – Qemu (Host) – Linux VM (Guest)

Workloads: Boot Linux kernel + UnixBench

High hit rate of HotTable
- 1% paths – 97% of exits
- 0.1% paths – 95% of exits
Increase HotTable Hit Rate

- Execution Pattern Inference: Hash = a pattern of similar executions
- Noise reduction (e.g., exclude interrupt handlers)
- Loop rerolling

**Execution trace** $i_1 i_2 i_3 \ldots i_m$  $\rightarrow$  **Pattern of executions** $bb_1 bb_2 \ldots bb_n$  $\rightarrow$  **Hash** $H_i$

*Noise reduction  
Loop rerolling*
Loop Rerolling Example

```c
a = 0;
for (i = 1..n)
    a = a + i;
return a;
```

| Paths | = 1 + |Range(n)|
Loop Rerolling Example

Solution: Loop rerolling
hShield Approach

- Workloads
- VM
- Hypervisor

Most popular paths

HotTable

- $H_1$
- $H_2$
- ...
- $H_t$

Hash $H_i$

Find $H_i$ in HotTable

Yes

Recomputed $H_i$ in CFG

Yes

Bad

No

Good

Profiling

- Small size
- Fast Lookup
- High hit rate (>95%)

Runtime checking

1. VM-exit
2. 1
3. 2
4. 3
5. 5
6. 6
7. VM-entry
hShield Approach

**Workloads**

**Hypervisor**

**HotTable**

Most popular

How to efficiently verify $H_i$ is recomputable in the **CFG**?

1. **Hash $H_i$**

2. Find $H_i$ in **HotTable**

   a. Yes
   
   b. No

3. Recompute $H_i$ in **CFG**

   a. Yes
   
   b. No

   i. Bad
   
   ii. **Good**

4. **Profiling**

5. **Runtime checking**

6. **VM-exit**

   1. 2
   
   3
   
   5
   
   6

**VM-entry**
Execution Path Reconstruction

- Input Hash $H_i$ and CFG $G = \langle V, E \rangle$
- Question: Exist a path $P \in G: Hash(P) = H_i$

- Naïve solution: Traverse $G$ until $P$ is found - impractical

- **hShield Solution:**
  - Using incremental hashing to efficiently **reconstruct** $P$ from $H_i$ and $G$
Execution Path Reconstruction

Is 1 first basic block?
Yes, $update(1, H_i) == H_i$

Is 2 first basic block?
Yes, $update(2, H_i) == H_i$

Is 3 first basic block?
No, $update(3, H_i) != H_i$

Is 4 first basic block?
Yes, $update(4, H_i) == H_i$

…. end at 6
Execution Representation

\[ f : \text{Exe} \rightarrow \text{Range} \]

- \( E \in \text{Exe} : E = I_1 I_2 \ldots I_n \)
  - \( I_i \): Instruction byte code (e.g., x86)

- \( \text{Range} \): Fixed length output

- \( f \) requirements
  - Collision resistant
  - Interactive – online construction
  - Incremental – online update
  - Facilitate loop rerolling implementation
Incremental Collision-free Hashing [1]

- **Randomization** – Derived from standard cryptographic functions (e.g., SHA, MD5)
- **Combination** – Algebraic operation
  - **Incrementality**: Allow update results when a portion of input changed without re-computing from scratch
- **Collision-free** [1]

hShield Counter Hash Function

Execution E

\[ \text{B'}_1=<1>.s.B_1 \quad \text{B'}_2=<2>.s.B_2 \quad \ldots \quad \text{B'}_n=<n>.s.B_n \]

\[ y_2 = \text{sha1}(\text{salt}.<2>.b_2) \]

\[ \text{MuHASH}(y_1, y_2) = y_1 \otimes y_2 \]
hShield Counter Hash Function

Execution $E$

- $B' = \langle 1 \rangle \cdot s \cdot B_1$
- $B'_{n} = \langle n \rangle \cdot s \cdot B_n$

- $\text{sha1}(B'_1)$
- $\text{sha1}(B'_2)$
- $\text{sha1}(B'_n)$

$F(E) = \otimes_{i=1..n}(\text{sha1}(\langle i \rangle \cdot s \cdot B_i))$

- $B = $ Basic block – facilitate loop rerolling
- $S: $ salt – individualize target
- $\otimes: $ modular multiplication (MuHash)
Security Evaluation

### Attack Model
- Change execution $E \rightarrow E' : f(E') \in F(E)$

### Solution
- Find $E'$ complexity = Discrete Log problem (assuming $h$ is ideal)
- Must harder to find an $E'$ which is valid x86 code
Scopes

- **Focus on design of the monitoring framework**
  - What/Where/How to monitor (*Answers: VM Exit / Hardware / Whitelist*)

- **Design for flexible future hardware implementation**
  - Make best effort to conduct measurement on actual hardware
    - *E.g., obtained supporting data on physical systems.*
  - Make best effort to anticipate problems in actual hardware implementation.
    - *E.g., issues with speculative execution, memory size constraints.*
  - Assume that we can place hooks in some basic signals in the hardware.
    - *E.g., intercept all interrupts and exceptions.*

- **Prototype the proposal in QEMU**
  - Software emulation of x86 processors and many external devices.
More to come...

- Architectural Design
- Evaluation with real attacks
- Performance evaluation