Keeping an Eye on Virtualization: VM Monitoring Techniques & Applications

Virtual Machine Introspection Overview
Overview

- What is virtual machine introspection (VMI)?
- VMI architecture
- Types of VMI
- Drawbacks of VMI
What is virtual machine introspection (VMI)?

- First proposed in 2003 by Garfinkel and Rosenblum
  - “A Virtual Machine Introspection Based Architecture for Intrusion Detection”
- VMI proposed to address the dichotomy between existing intrusion detection systems
VMI Design Goals

- VMI leverages virtualization in three ways:
  - Isolation
  - Inspection
  - Interposition

- Use these features to provide security and reliability for VMs
  - We will focus on security
Typical VMI architecture
Typical VMI architecture
Typical VMI architecture
Typical VMI architecture
Passive VMI

- Relies on polling VM for state information
  - Typically rely on whitelisting/blacklisting
  - Our example: LibVMI
  - Other examples: Livewire, Virtual Inspection on Xen (VIX)
Passive VMI

Check this on some interval
Active VMI

- Enabled by interposition
- Relies on responding to events
  - Look for anomalous events
  - Event tracking
  - Our examples: Hypertap/Hprobes
  - Other examples: Livewire, Lares, Sim, Antfarm, Lycosid
Active VMI
Drawbacks of VMI

- **Performance overhead**
  - VM is paused when checks are performed
- **Semantic gap makes it hard to extract actionable information**
- **Relies on isolation of VMs from hypervisor**
  - Hypervisor bugs can lead to compromise
  - See: Cloudburst and Xen hypercall bugs
(Passive) LibVMI Tutorial
Overview

- What is LibVMI?
- How does LibVMI work?
- Example LibVMI monitor (demo)
- Weaknesses of passive VMI monitoring
What is LibVMI?

- Library for implementing VMI monitors with a focus on memory introspection
- Grew out of XenAccess, was developed at Sandia National Laboratories, and released open source
- Tries to simplify the process of writing VMI monitors
What is LibVMI’s platform?

- Hypervisors supported:
  - KVM
  - Xen

- Guest OSes supported:
  - Windows
  - Linux

- Aims to be extensible
Components and Architecture

LibVMI System
(Host OS/Privileged VM)

- Volatility
- App (Python)

App (C)
PyVMI
LibVMI

Read/Write Memory
Hardware Events
Address Translation
Symbol Resolution

Monitored VM(s)
- Windows (full support)
- Linux (full support)
- Other OS (partial support)

Hypervisor / Virtualization
- Xen
- KVM
- QEMU
- PhysMem in File

[4]
Why is LibVMI useful?

- Powerful abstraction
  - Fast prototyping
- Capable of passive and active monitoring
  - Active monitoring limited to Xen
- Caching leads to good performance
- No modification of guest OS
Example LibVMI Use Cases

- Process listing
- Kernel module listing
- Get open sockets (we will demonstrate this in Linux)
- Watch for syscalls
- View memory pages
- So much more!
Things of note

- LibVMI needs additional information provided by a configuration file
  - Example: offsets of kernel data structures
- Applications on KVM host do not have access to guest VM memory
  - Either patch KVM or use remote debugging via GDB
KVM memory access
LibVMI reading kernel symbol

1. VMI app requests to view kernel symbol (e.g., current)
LibVMI reading kernel symbol

2. LibVMI finds the virtual address of symbol by via supplied configuration information
LibVMI reading kernel symbol

3. Page directory mapped to find correct page table
LibVMI reading kernel symbol

4. Page table mapped to find correct page
LibVMI reading kernel symbol

5. Correct data page returned to LibVMI
LibVMI reading kernel symbol

6. LibVMI returns the data to the app (or continues to map pages)
What goes on within LibVMI?

- `vmi_read()` is a core function that facilitates reading memory from the guest VM
- Let’s take a look at the call graph of `vmi_read()` to get a better understanding of what goes on under the hood of LibVMI
What goes on within LibVMI?

- It does a lot!
  - This means you have to do less when using it
- Let’s take a look at groups of functions
LibVMI Caching

- LibVMI utilizes two caches to improve performance
  - Page-level cache
  - Virtual to physical address translation cache
- They can be disabled if you so choose
Page-level cache

VMI Application

Read Memory

In Cache?

Yes

No

Buffer

Notify

Page Cache LRU List

Fresh

Stale

if (lru is full)
remove 1/2 most stale

Page Cache Hash Table

Hash-1 → Handle/Buf-A
Hash-2 → Handle/Buf-B
Hash-3 → Handle/Buf-C
...
Hash-n → Handle/Buf-n

Hash = Paddr

Memory Request

LibVMi

Handle or Buffer

Hypervisor / VMM

[7]
Virtual to physical cache

![Diagram of virtual to physical cache]

- VMI Application
- Translate Vaddr
- Paddr
- In Cache?
  - Yes
  - Valid?
    - Yes
    - V2P Cache
      - Hash-1 ➔ Handle/Buf-A
      - Hash-2 ➔ Handle/Buf-B
      - Hash-3 ➔ Handle/Buf-C
      - [Hash-n] ➔ Handle/Buf-n
    - No
      - V2P Cache
        - Hash-n ➔ Handle/Buf-n
  - No
    - Walk Guest Page Tables
    - Paddr
    - Memory Reads
    - Hypervisor / VMM

Hash = CityHash(va << 64 | cr3)
Cache Performance

System configuration: Xen 4.1.1, Dual Intel Xeon X5675, 24G RAM, Windows XP VM
Times shown are for cache hits, when possible

- vmi_translate_ksym2v
- vmi_translate_kv2p

No Cache: vmi_translate_ksym2v = 1331, vmi_translate_kv2p = 50
Page Only: vmi_translate_ksym2v = 123, vmi_translate_kv2p = 6
Addr Only: vmi_translate_ksym2v = 33, vmi_translate_kv2p = 6
All Cache: vmi_translate_ksym2v = 6, vmi_translate_kv2p = 6
Example passive LibVMI monitor skeleton

- Initialize VMI instance
  - vmi_init
- Pause VM for consistent memory access
  - vmi_pause_vm
- Read memory from VM (introspection step)
  - vmi_read
- Resume VM
  - vmi_resume_vm
- Clean up VMI instance
  - vmi_destroy
Setup VMI

vmi_init

vmi_pause_vm

Introspection

vmi_read

Tear Down VMI

vmi_resume_vm

vmi_destroy
Socket Listing LibVMI Demo Overview

- Capabilities
- How does it work?
- Why would this be used?
- Monitor details
- Demo time!
- Implementation details
Socket listing LibVMI Demo Capabilities

- Extend included process listing example to access file info for processes
- Check if any of the files is a socket
- Includes both Unix and TCP sockets
Linux Kernel Data Structures

- Process listing example walks `task_struct list`
- Each `task_struct` contains information related to a process
Linux Kernel Data Structures

- Each processes’ `task_struct` contains file information
- Follow file related data structures and determine which files are sockets
Why would you use this?

- Have whitelist of processes that can have network sockets
- If process detected by monitor is not on whitelist, alert administrator
Monitor architecture and components

LibVMI System
(Host OS/Privileged VM)

- Volatility
- App (Python)
- App (C)
- PyVMI
- LibVMI

Monitored VM(s)

- Linux (full support)
- ... (partial support)

Hypervisor / Virtualization
- Xen
- KVM
- QEMU
- PhysMem in File

Read/Write Memory
Hardware Events
Address Translation
Symbol Resolution

[4]
Example LibVMI configuration

HOST_MACHINE: # cat /etc/libvmi.conf
ubuntuvm1 {
    ostype = "Linux";
    sysmap = "/boot/System.map-3.11.0-17-generic";
    linux_name = 0x304;
    linux_tasks = 0x1d8;
    linux_mm = 0x1f4;
    linux_pid = 0x228;
    linux_pgd = 0x20;
}

ubuntuvm2 {
    ostype = "Linux";
    sysmap = "/boot/System.map-3.11.0-17-generic";
    linux_name = 0x304;
    linux_tasks = 0x1d8;
    linux_mm = 0x1f4;
    linux_pid = 0x228;
    linux_pgd = 0x20;
}
Monitor output

[ 1] init (struct addr:c6880000)
   init has 2 sockets
[ 317] udevd (struct addr:c7b20ce0)
   udevd has 2 sockets
[ 353] rsyslogd (struct addr:c7bcda20)
   rsyslogd has 1 sockets
[ 502] sshd (struct addr:c43fe700)
   sshd has 2 sockets
[ 681] sshd (struct addr:c43fcd40)
   sshd has 3 sockets
[ 690] sudo (struct addr:c45d6700)
   sudo has 1 sockets
[ 7514] apache2 (struct addr:c021b380)
   apache2 has 1 sockets
Demo Time!
Monitor implementation details

- Closely follows example skeleton shown previously
  - Setup and tear down are the same
- Utilize following functions for introspection:
  - `vmi_translate_ksym2v` (**find** `init_task`)
  - `vmi_read_str_va`
  - `vmi_read_32_va`
  - 7 calls to `vmi_read_addr_va`
- Seems pretty easy, right?
Weaknesses of passive VMI monitoring

What is wrong with this picture?
Weaknesses of passive VMI monitoring

- **Passive polling-based monitoring is vulnerable to transient attacks**
  - Malicious activity occurring between monitoring checks are not detected
Weaknesses of passive VMI monitoring

Observer notes disruption and infers that there is hypervisor activity.

Hypervisor pauses VM and checks VM state.

Observer

VM

Hypervisor

Beacon signal over network

System active/running

VM suspend time side channel

Time
VM suspend side-channel

- Network based (previous slide)
- Local guest OS based
  - Hook into various functions and time call frequency
Example attack scenarios

- Invisible (large) file transfer
- “Persistent” backdoor
  - Constantly connect/disconnect to avoid being detected
Summary

- VMI aims to provide low risk high fidelity monitoring without sacrificing security
- LibVMI is a library for simplifying writing VMI monitors
- Passive monitoring has some inherent weaknesses
Resources and more information

- **New website:** [http://libvmi.com/](http://libvmi.com/)
- **Doxygen:** [http://libvmi-kvm.gtisc.gatech.edu/index.html](http://libvmi-kvm.gtisc.gatech.edu/index.html)
  - Good for call graphs and function documentation
- **GitHub:** [https://github.com/libvmi/libvmi](https://github.com/libvmi/libvmi)
- **Sandia report:** [http://prod.sandia.gov/techlib/access-control.cgi/2012/127818.pdf](http://prod.sandia.gov/techlib/access-control.cgi/2012/127818.pdf)
- **A good overview of LibVMI from the author**
Image sources

- [6] http://libvmi-kvm.gtisc.gatech.edu/libvmi_8h.html#a2ade5f3ee0a27364496709a0c2ef1521
- [8] http://www.makelinux.net/books/lkd2/ch03lev1sec1