Evasion Resistant Intrusion Detection Framework at Hypervisor Layer in Cloud

Bhavesh Borisaniya and Dr. Dhiren Patel
NIT Surat, India
Email: {borisaniyabhavesh, dhiren29p}@gmail.com

Abstract—Virtual Machine Introspection (VMI) describes the method for monitoring and analyzing the state of a virtual machine from the hypervisor layer. VMI can be used to detect/prevent the malicious use of cloud infrastructure. To implement any VMI based application, one has to bridge the semantic gap problem. The challenge is to make sure that system state monitoring component itself should not be prone to attack. In this paper, we propose, an evasion resistant VMI based intrusion detection framework utilizing virtual machine hardware knowledge (about virtual CPU, virtual RAM, virtual storage and virtual NIC) to detect in-VM malicious activities.

Index Terms—Virtual machine introspection (VMI), Intrusion Detection System (IDS), Cloud Computing, Infrastructure as a Service (IaaS)

I. INTRODUCTION

Virtualization introduces a software abstraction layer between the hardware and the operating system and applications running on top of it [1]. This abstraction layer, called virtual machine monitor (VMM) or hypervisor, hides the physical resources of the computing system from the operating system (OS). VMM are of two types: Type I and Type II VMM as shown in Figure 1. A Type I VMM runs directly on the physical hardware, also called as native or baremetal (for example, VMware ESXi and Citrix Xen server). A Type II VMM runs as an application in a normal operating system (for example, VMware Workstation, VirtualBox, etc.). Virtualization makes it possible to run multiple operating systems in parallel on the same hardware. The hardware platform is partitioned into one or more logical units called virtual machines (VMs). Two primary security benefits offered by virtualization are resource sharing and isolation.

A virtual machine does not have the direct access of the host hardware. All the guest OS privileged operations are trapped by VMM. The system call trapping mechanism for Type II is more expensive than Type I VMM. As shown in Figure 2, in Type I case, to support privileged operation, four mode switches (guest app – hypervisor – guest OS – hypervisor – guest app) are required. While in Type II case, trapping requires eight mode switches (guest app – host OS – hypervisor – host OS – guest OS – host OS – hypervisor – host OS – guest app).

Virtualization has a key role in supporting the IaaS service model of cloud computing. One can achieve multi-tenancy in cloud by sharing physical resources among multiple clients (tenants). Free trial and cheap rental of cloud services draw attackers to host their malicious code on cloud infrastructure. Attackers then use it to infect other machines in the cloud and/or to external network. They take the advantage of cloud
scalability and availability for uninterrupted attack on victim, and use cloud firewall for self-protection. Cloud used for such malicious purpose is called Dark Cloud [2]. Such dark cloud has the capabilities to bring down the business services through collaborative attacks such as flooding and distributed denial of service attack (DDoS) [2]. It is the responsibility of cloud provider to provide secure environment to client. Typically, cloud provider monitors the cloud infrastructure by introducing agent inside each virtual machine. However, this agent itself can be attacked and compromised by an attacker. Cloud provider cannot rely upon traditional in-guest security, because if one virtual machine is compromised, then it can be used to attack other virtual machines.

The virtualization layer in cloud adds new attack surface for attackers and therefore there are new challenges to prevent attacks at this layer. All hosted virtual machines are managed by hypervisor and therefore it is a central point of attack to acquire control. Virtual machine based rootkit and malware can exploit hypervisor vulnerability to gain access of hosted virtual machines. VM escape is a kind of virtualization attack in which program running in virtual machine is able to break the isolation provided by virtual machine monitor (VMM) and get access to the host machine. In 2008, vulnerability (CVE-2008-0923) in VMware discovered by Core Security Technologies proved that VM escape is possible [3]. Recently, a similar vulnerability (CVE-2012-0217) in Xen 4.1.2 is reported. This vulnerability affects the way Intel processors implement error handling in the SYSRET instruction. This flaw allows local users to gain administrative privileges via a crafted application [4]. In order to monitor such in-VM malicious activities from outside, VMI could be a good solution from cloud provider’s perspective, as it may not require to put any agent inside the VM for monitoring. VMI monitors and analyzes the system state with hardware state (i.e. CPU and I/O register, system calls etc.) and software state (i.e. VM physical memory and hard disk content) of running VM to get the inside view. The generated view can be used for detection/prevention of any malicious activity at hypervisor layer.

We propose a security framework (using VMI technique) to detect in-VM malicious activities from outside of VM. It works at hypervisor layer and monitors hardware state of VM to analyze intention of various activities performed by the VM.
The rest of the paper is organized as follows: Section 2 describes the VMI and its implementation methods. Related approaches are discussed in section 3. Proposed security framework is discussed in section 4 with conclusion and references at the end.

II. TECHNICAL BACKGROUND

A. Virtual Machine Introspection (VMI)

Virtual machine introspection is a technique to monitor inside view of the VM by analyzing its machine state information from outside of VM [5]. The key challenge with the VMI is to bridge the view difference of information inside the VM (i.e., processes, files etc.) and outside the VM (i.e., CPU register value, system calls, I/O request, etc.). This difference is known as semantic gap [6]. For example as shown in Figure 3 from inside the VM we can get list of running processes by command `ps`. Same information can be retrieved by processing memory dump of VM, which is in terms of low level bytes.

Figure 3. Semantic Gap in VMI

Pfoh et al. [5] have presented three view-generation patterns to bridge this gap as follows (Figure 4):

**In-band delivery:** The in-band delivery pattern describes an approach in which an agent residing inside the guest machine creates a view and delivers it to the VMM [5]. It uses the guest OS's inherent knowledge of the software architecture. As it resides inside the guest OS it has broad scope of viewing the exact information. However, the attacker can compromise in-guest agent and can send false information to VMI application. Also, through this method, VM state information cannot be collected if virtual machine is in suspended state. This pattern can be useful for second opinion for information reliability similar to lie detection. As this method is dependent on semantic knowledge of guest OS, it is vulnerable to Direct Kernel Structure Manipulation (DKSM) [7]. DKSM is an attack, which can effectively subvert and confound existing VM introspection tools. It compromises a guest such that the kernel’s use of any field of its data structures (or templates) could be potentially modified [7].

**Out-of-band delivery:** The out-of-band delivery pattern describes an approach in which system state is monitored without getting into guest machine. Here, semantic knowledge is delivered by an external function [5]. For example, the virtual machine monitor (VMM) may make use of a previously delivered symbol table (based on the guest OS kernel) to determine the position of key data structure. As out-of-band approach is not bound to the current state of the monitored machine [5]. It can be used with suspended virtual machine. However, it is not possible to use out-of-band delivery if the layout of the system state changes during introspection. An attacker could change the kernel data structure to fool the VMI application [7].

**Derivation:** Derivation is a pattern that derives information through semantic knowledge of the hardware architecture [5]. Access to the hardware state is provided by the VMM. It can monitor CPU’s register or can
analyze system call execution to monitor processor raised software interrupt [5]. The state information is completely derived from the current hardware state. An attacker cannot hide its presence from the underlying hardware. Derivation is guest OS portable, reliable and tamper resistant. It has very constrained view due to limited information extracted by monitoring hardware states only [5].

B. VMI Implementation Methods

There are various methods to implement the VMI as follows [8][9]:

**VM State Access:** VM State Access is the method by which VMI application can get VM state information through VMM. A VM state is defined by its CPU registers, memory space and I/O access of that VM [5][8]. The VMM can provide low level access to the VM state. It is the only passive method for VMI. VMI application that uses this method is responsible for providing semantic knowledge and intelligence about guest OS. This is useful to translate the low level view of guest VM in meaningful information.

**Guest OS Hooks:** Guest OS Hook is a VMI implementation method that uses in-VM agent to generate the system view. Hooks are kernel modules inserted into the guest OS. They send back the required information to VMI application [9]. Guest OS hooks are more intrusive and require guest OS modification. This method is used for active monitoring and for preventing compromises.

**Interrupts:** Interrupt method utilizes the general system interrupts (such as page faults, invalid opcode etc.) and context switching interrupts which are available in native virtualization to facilitate VMI [8]. Also, debugging exception interrupt allow hardware breakpoints, which are supported by the built in debugging mechanism available in modern processors [8][9]. Using this interrupts, processor generate debug exception when instruction pointer (IP) register hit addresses defined in debug address registers.

**Kernel Debugging:** Kernel debugging works by inserting breakpoint opcode in arbitrary memory addresses [9]. This type of breakpoint is called software breakpoint. It is like debugging exceptions in CPU architecture, except that instead of being implemented in CPU; it is a part of VMM. In contrast to debug exception and hardware breakpoints, more functionality can be provided in this method.

III. RELATED APPROACHES

In order to bridge the semantic gap, many VMI based approaches are proposed in literature, which inherently uses VMI libraries for gathering system state information. These VMI libraries either uses one of the delivery pattern (i.e. in-band or out-of-band) or derivation based approach for view generation that is further used for building VMI based security framework (i.e. IDS or IPS). Table 1 shows the summary of existing VMI libraries.

Amani S Ibrahim et al. [10] have used VMsafe to build CloudSec, a virtualization aware monitoring appliance. It provides active, transparent and real-time security monitoring for hosted VMs in the Cloud IaaS model. It actively reconstructs and monitors the dynamically changing kernel data structures instances for detection and prevention of kernel data rootkits. However, VMsafe library, which is the base of this concept is not freely available and can be used with only VMware hypervisor. VMwall [11] is an application level...
firewall and built upon Xen hypervisor. It uses XenAccess as VMI component for analyzing the VM physical memory. Ether [12] is designed for malware analysis and based on hardware virtualization extensions like Intel-VT. It resides outside of the target OS and works on Xen hypervisor. In order to monitor system interrupt, it forces the debug exception to occur after execution of every instruction by setting trap flag. It does not support trapping of user interrupts. It is performance intensive as it creates a debug exception after every instruction. This library is not active and has not been updated for some time as of this writing. HyperSLueth [13] is designed for live forensic analysis, uses the similar approach as Ether for system call tracing of potentially compromised production system.

Apart from these approaches, there are other VMI based intrusion detection/prevention approaches, which are having their own mechanism for bridging the semantic gap. Livewire [14] is considered as the first VMI based IDS. It depends on both in-band and out-of-band delivered information for view generation. It requires the introspecting component to be located within the host machine [15]. To interpret the VM’s machine state, it requires the guest OS knowledge. HyperSpector [16] is another VMI based IDS. It places the introspecting component inside a second VM running on the same host as the monitored VM to leverage the isolation provided by the VMM. However, it is limited to passive system checks, which makes event-based monitoring extremely difficult [17]. VMwatcher [18] is the out-of-the-box intrusion detection system in VMI based environment. It does introspection on files and memory and supports multiple VMMs. It does file introspection by mounting the guest’s hard disk into the introspecting machine and memory introspection by a technique called Guest View Casting [18]. Livewire, VMwatcher and VMwall use external guest OS kernel

<table>
<thead>
<tr>
<th>VMI Library</th>
<th>Supported Hypervisor</th>
<th>Supported Guest OS</th>
<th>View Gen. Pattern</th>
<th>Monitoring</th>
<th>Method</th>
<th>Characteristics</th>
</tr>
</thead>
</table>
• The project is no longer being maintained and evolved into LibVMI. |
• It reads/writes memory, provides APIs for accessing CPU registers, pausing and unparsing a VM etc.  
• Uses in-guest agent to create introspection program that can extract required information from outside of VM.  
• The introspection program generated is guest OS specific and thus vulnerable to DKS trip attack [7]. |
| Virtuoso [21] Open Source | KVM | Windows, Linux, Haiku | In-band, Out-of-band | Active | Guest OS Hooking | • VMsafe APIs allows vendors to build security products, which can detect and stop advanced malware.  
• It works with VMware hypervisor and currently available to VMware partners only. |
| VMWare VMsafe [22] Proprietary | VMware ESXi(i) | Linux, Windows | Unknown | Unknown | Unknown | • Can work with memory dump file or with read access of VM physical memory. It will support Windows in future.  
• It provides interactive shell for memory analysis |
| InSight [23] Open Source | Xen, KVM | Linux | Out-of-band | Passive | State Access | • It can work with memory dump file or with read access of VM physical memory. It will support Windows in future.  
• It provides interactive shell for memory analysis |
| Nitro [24] Open Source | KVM | Linux, Windows | Derivation | Active | Interrupt | • It extends the KVM to support hardware based system call trapping for VMI applications.  
• It supports all three system call mechanisms provided by the Intel x86 architecture. It also supports user interrupts.  
• It monitors memory writes by using shadow page tables and privilege over the guest in handling page faults. |
| Ether [25] Open Source | Xen | Linux, Windows | Derivation | Active | Kernel Debugging | • It is designed for malware analysis.  
• To monitor system interrupt it forces the debug exception to occur after execution of every instruction by setting trap flag. It does not support user interrupts.  
• It monitors memory writes by using shadow page tables and privilege over the guest in handling page faults. |
data structure information to reconstruct the semantics of what is executing within the guest and are thus vulnerable to the DKSM attack [7].

In contrast to above systems, IntroVirt [26] and KvmSec [27] are VMI based intrusion prevention system capable of preventing attacks. IntroVirt provides a way to delay security patches while still prevent zero day attacks by allowing administrator to define vulnerability predicates. However, vulnerability predicates need to be written manually and require vendor co-operation for closed-source software (in order to write correct predicate). KvmSec is designed to protect guest VM against viruses and kernel rootkits. It relies on installing security code inside the guest VMs to obtain high-level OS abstractions from inside the guest. Security code installed inside the guest VM is not reliable. If VM is compromised, it can send false information to protection framework [10]. Table 2 shows the summary of these approaches.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Type</th>
<th>Characteristic(s)</th>
<th>Limitation(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Livewire [14]</td>
<td>VMI based</td>
<td>• Enforce security policy on guest virtual machines</td>
<td>• It lacks of the isolation provided by a separated VM</td>
</tr>
<tr>
<td></td>
<td>IDS</td>
<td>• Depends on both in-band and out-of-band delivered information</td>
<td>• Vulnerable to DKSM attack</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Introspecting components are located within the host operating system</td>
<td></td>
</tr>
<tr>
<td>HyperSpector</td>
<td>VMI based</td>
<td>• Place the introspecting component inside a second VM running on the same host as the monitored VM</td>
<td>• Limited to passive system checks, which makes event-based monitoring extremely difficult</td>
</tr>
<tr>
<td>[16]</td>
<td>IDS</td>
<td>• It provides three inter-VM monitoring mechanisms: software port mirroring, inter-VM disk mounting, and inter-VM process mapping.</td>
<td></td>
</tr>
<tr>
<td>VMwatcher</td>
<td>VMI based</td>
<td>• It does introspection on files and memory File introspection can be done by mounting the guest’s harddisk into the introspecting machine</td>
<td>• Vulnerable to DKSM attack</td>
</tr>
<tr>
<td>[18]</td>
<td>IDS</td>
<td>• Memory introspection can be done by Guest View Caster</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• It is application level firewall and built upon the XEN</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Uses the XenAccess to introspect the monitored machines physical memory</td>
<td>Vulnerable to DKSM attack</td>
</tr>
<tr>
<td>VMwall [11]</td>
<td>VMI based</td>
<td>• It can be installed in the system on-the-fly, without reboot</td>
<td></td>
</tr>
<tr>
<td></td>
<td>firewall</td>
<td>• Three applications developed top of it – a lazy physical memory dumper, a lie detector, and a system call tracer</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Uses a similar approach as Ether for system call tracing</td>
<td></td>
</tr>
<tr>
<td>CloudSec</td>
<td>Virtualization aware monitoring appliance</td>
<td>• Provides active, transparent and real-time security monitoring for hosted VMs in the IaaS model</td>
<td>Uses VMsafe library, which is not freely available</td>
</tr>
<tr>
<td>[10]</td>
<td></td>
<td>• It actively reconstructs and monitors the dynamically changing kernel data structures instances for detection and prevention of kernel data rootkits</td>
<td></td>
</tr>
<tr>
<td>HyperSlueth</td>
<td>Live forensic analysis framework</td>
<td>• It can be installed in the system on-the-fly, without reboot</td>
<td>Performance intensive like Ether.</td>
</tr>
<tr>
<td>[13]</td>
<td></td>
<td>• Uses a similar approach as Ether for system call tracing</td>
<td>• Does not support user interrupts.</td>
</tr>
<tr>
<td>IntroVirt</td>
<td>VMI based</td>
<td>• Provides a way to delaying security patches while still prevent zero day attacks by allowing administrator to define vulnerability predicates</td>
<td>Vulnerability predicates need to be written manually</td>
</tr>
<tr>
<td>[26]</td>
<td>IPS</td>
<td>• Works with VM snapshots, replay old executions and monitors current system execution</td>
<td>• Require vendor cooperation for closed-source software</td>
</tr>
<tr>
<td>KvmSec</td>
<td>VMI based</td>
<td>• It can protect guest virtual machines against attacks such as viruses and kernel rootkits</td>
<td>Security code installed inside the guest VM is not reliable. If VM is compromised it can send false information.</td>
</tr>
<tr>
<td>[27]</td>
<td>IPS</td>
<td>• Relies on installing security code inside the guest VMs to obtain high-level OS abstractions from inside the guest</td>
<td></td>
</tr>
</tbody>
</table>

IV. PROPOSED APPROACH

Our goal is to build an evasion resistant VMI based intrusion detection framework capable of detecting in-VM activity from outside of VM at hypervisor layer. As shown in Figure 5, our framework has two main components: h/w based system call tracing tool and out-of-hand delivery pattern based VMI library. System call tracing tool will be used for VM system call tracing, while VMI library is used for analyzing the VM memory.

H/W based system call tracing tool runs at hypervisor layer and extracts required system calls for detecting intrusion in VM. The idea behind using the system calls for VMI is its evasion resistant nature; it follows derivation method and uses only hardware knowledge for view generation. It takes the advantage of Intel/AMD VT (Virtualization Technology) to trap the system calls. The task of this component is to collect the required system call information of monitored VM and deliver it to analyzer for further analysis. From
the literature, we found that Nitro [24] is suitable for this task, as it supports all three system call mechanisms (Interrupt-based, SYSCALL-based and SYSENTER-based) provided by Intel x86.

As mentioned earlier; out-of-band delivery pattern based VMI library require the guest OS kernel data structure information (as a semantic knowledge) for interpreting the VM memory content. Approaches using out-of-band delivery based VMI library are vulnerable to DKSM attack [6]. However, in our proposed framework, the VMI library plays a supportive role in context with the system call tracing tool. It is used for extracting details from VM memory relevant to system calls traced by the system call tracing tool. We use InSight and/or LibVMI for this purpose. The VMI library component uses guest OS kernel data structure information in terms of symbol table (System.map file in case of Linux OS) as semantic knowledge for interpreting memory contents. Cloud IaaS service model can deliver VM with different preinstalled guest OS to clients. Hence, it will be good idea to extract kernel symbol table of each OS before deploying it on cloud and store it at safe place for later use. While extracting kernel symbol table from each OS, extra care should be taken in conforming that no malware is installed/running in it. Finally, this extracted kernel symbol table will be used by VMI Library to analyze VM memory. Then it delivers this analyzed information to analyzer component to detect intrusions.

![Proposed Framework for one node](image)

By co-relating information delivered by VMI library with system call information delivered by system call tracing tool and matching it with precompiled intrusion profiles, analyzer component will take classify activity as either intrusion or normal activity. If it identifies the activity as an intrusion it will raise an alert through alert generation component. Intrusion profiles are behavior profiles having rules (in terms of collection of system calls) which can distinguish intrusion behavior from normal one. The key challenge in this is to define the intrusion profile which can accurately classify intrusion and normal activity from collected system call information. In order to build such intrusion profiles, suitable machine learning technique should be used. We are currently exploring suitable technique to fulfill this task. However, we believe that the information extracted by system call tracing tool (Nitro), is sufficient for machine learning technique to classify intrusion and normal behavior.

For integrating this approach in Cloud, one of the possible architectures is shown in Fig 6. Here, h/w based system call tracing tool (i.e. Nitro) will be placed in each node controller. Analyzer component, which is residing along with Nitro in each node controller, fetches intrusion profile from central database.

A. Analysis

System call tracing tool used in our proposed framework uses derivation method for VMI. Derivation method uses VM hardware knowledge for view generation. Any malware has to go by the rules of VM hardware, i.e., it cannot change the behavior of hardware. Thus, malware running in VM cannot circumvent out-of-the-box derivation based VMI. Also, in case of complex analysis, to get deeper information of system calls, memory analysis will be done using out-of-band VMI library. Its role is to supplement the knowledge acquired by system call tracing tool. As it derives the properties of derivation method, our framework is evasion resistant against in-VM attacks.
Our framework uses behavioral knowledge to classify the intrusions. We can record the normal behavior of the system and define the potential intrusion as deviation from normal behavior. Or we can create the intrusion profiles of abnormal behavior and match it with the current behavior. As creating intrusion/normal behavior profiles is important and time consuming task, we are exploring suitable technique to automate it.

V. CONCLUSIONS

By utilizing virtual machine introspection, cloud provider can monitor the VM activity without any in-VM client component. Derivation method used in VMI is known to be evasion resistant as it uses hardware knowledge for view generation. Though, out-of-band delivery pattern are prone to kernel data structure manipulation, it has wider view and can support derivation based approaches for deeper analysis. In the proposed VMI based intrusion detection framework, we used system call tracing tool to acquire virtual machine state information. The information collected is analyzed through out-of-band VMI library to detect in-VM malicious activity. According to the level of information provided by system call tracing tool, machine learning technique can be used to build normal/intrusion profiles.

In this paper, we reported the state-of-the art of VMI and identified obstacles in building the VMI applications. Our proposed framework is evasion resistant under the guarantees provided by derivation based approach for view generation. It is under development and being explored with cloud, results are encouraging.