AN ARCHITECTURE FOR TRUSTWORTHY SERVICES BUILT ON EVENT BASED PROBING OF UNTRUSTED GUESTS

BY

READ SPRABERY

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Advisers:

Professor Rakesh Bobba
Professor Roy Campbell

ABSTRACT

Numerous event-based probing methods exist for cloud computing environments allowing a trusted hypervisor to gain insight into guest activities. Such event based probing has been shown to be useful for detecting attacks, system hangs through watchdogs, and also for inserting exploit detectors before a system can be patched, among others. In this paper, we illustrate how to use such probing for trustworthy logging and highlight some of the challenges that existing event based probing mechanisms do not address. These challenges include ensuring a probe inserted at given address is trustworthy despite the lack of attestation available for probes that have been inserted dynamically. We show how probes can be inserted to ensure proper logging of every invocation of a probed instruction. When combined with attested boot of the hypervisor and guest machines, we can ensure the output stream of monitored events is trustworthy. Using these techniques we build a trustworthy log of certain guest-system-call events powering a cloud-tuned Intrusion Detection System (IDS). Additionally, we identify new types of events that must be added to existing probing systems to ensure attempts to circumvent probes within the guest appear in the log. We highlight the overhead penalties paid by guests to ensure log completeness when faced with probabilistic attacks and show promising results (less that 10% for guests) when a guest is willing to relax the trade-off between log completeness and overhead. Our demonstrative IDS shows the ability to detect common attack scenarios with simple policies built using our guest behavior recording system.

To my wife, Brittany, for her love and support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

Acronyms

BMC Base Management Controller

EPT Extended Page Tables

HAV Hardware Assisted Virtualization

IDS Intrusion Detection System

ISR Interrupt Service Routine

MAC Mandatory Access Control

MSR Model Specific Register

NPT Nested Page Tables

TDP two-dimensional page tables

VA virtual appliance

VM Virtual Machine

VMI Virtual Machine Introspection

VMM Virtual Machine Monitor

CHAPTER 1

INTRODUCTION

Cloud computing lends itself to service oriented architectures as one can more efficiently manage services that run as separate virtual machines. This approach has led to virtual machine images being sold in marketplaces as so called Virtual appliances (VAs) meant to run single services [1]. The second aspect of cloud computing that makes it more amenable to better IDS systems is the hypervisor's ability to inspect guest memory to provide new services [2, 3, 4, 5].

In previous work, we used the hypervisor as a basis for detecting rootkits [6]. Examples from literature have used the hypervisor to detect malware that hides itself from process-listing tools using a variety of approaches [7, 2]. Many of these approaches either require running a second VM [8, 9], rely heavily on knowledge about kernel data structures[10], or focus on specific types of intrusions or malware[11, 7, 2, 12]. Our approach aims to log malicious activity so higher level services can take action before a malicious actor has had the chance to modify kernel data structures in an effort to circumvent detection (e.g.: remove itself from the process list).

Our logging technique utilizes hypervisor level probes such as those presented by Estrada et al. [13] and Lengyel et al. [14]. These probing techniques utilize Hardware Assisted Virtualization (HAV) to allow the hypervisor to insert probes into guest memory by replacing an instruction with an instruction that causes control to transfer back to the hypervisor (through a `VMExit`). The hypervisor can then inspect guest memory before transferring control back to the guest. Our IDS is built around logs gathered using only probes placed at specifically chosen system calls. The system call interface tends to be very stable and only requires knowledge regarding which arguments are passed in which CPU registers. Specifically, we reduce the performance impact of a probe induced `VMExit`'s by evaluating the detection coverage with the minimal number of system calls probed. We build probes

that hook two system calls in Linux, `sys_exec` and `sys_open`, highlight the trade-off between performance and logging guarantees, and show an example of a service built on top of such a log in the form of an IDS. By hooking only certain system calls we aim to lower the performance penalty paid by guests (by logging less information) while increasing the costs to execute a successful attack against the guest (by limiting the actions that can be taken without being logged). We do not protect against every attack on the guest, but we aim to protect against attacks on the logging system that originates from the guest while increasing the burden of performing unloggable malicious activity.

In this paper we exploit the "appliance" nature of cloud computing to develop a service oriented IDS that is easy to manage, has few false positives, and is built upon a trustworthy logging service running inside of the hypervisor. We have implemented our system on Linux 3.13 (Ubuntu 14.04 LTS) as the KVM hypervisor host, and are able to detect intrusions into the popular blogging framework Wordpress. Additionally, we explore how event based probing systems can be loaded before the probed instruction(s) have the chance to execute even once. Probe insertion before execution is guaranteed by inducing a unique sequence of page faults in the hardware accelerated guest-physical-address to host-physical-address translation available in modern processors. By combining existing trusted boot techniques for both the hypervisor and guests, write protecting the probed instruction, and monitoring specific hardware registers, we can guarantee event log completeness. Note that currently we do not monitor other hardware generated events, such as those stemming from a Base Management Controller (BMC). We show that the integrity of the probe cannot be fully guaranteed by existing probe based monitoring system and that two more traps which transfer control back to the hypervisor must be added to ensure correctness of a system call based log. Our contributions include a methodology for building trustworthy services that must use data from untrustworthy guests and the identification of the events that must be logged to guarantee the integrity of any data driven response such a system may produce.

1.1 Goals of a Hypervisor-Based Trusted Log

We set forward four requirements that must be met to guarantee the integrity of a trusted log meant to monitor guest Virtual Machines (VM's). Again, while this logging does not prevent attacks on guests, it can reduce the number of attack-related events that go unlogged. Guaranteeing the integrity and completeness of our trusted log provides guarantees for future work in higher level services built using such a log. The requirements are as follows:

R1 Information provided by the guest cannot alter the logging entity's control flow. Information is simply logged and higher level services can respond to logged data appropriately,

R2 Guests cannot modify or remove an event from the log after the fact,

R3 In-guest modifications to instrumented locations should be logged,

R4 Modifications to functions invoking the hooked instruction should similarly be logged,

R5 The event log must contain every event ϵ_T of type T if there exists any probe P_T in the set of probes which produces output corresponding to events of type T, up to and including a malicious action within the guest.

We also have three design goals that drive the engineering choices behind the architecture proposed here. These are:

D1 Minimize the performance impact on guests,

D2 Minimize additions to the trusted compute base, and also

D3 Require no modification of guests (i.e., transparent to end users).

R1 implies that probes must not trust memory read from guests. In particular, probes must not trust the guest to inform the logging function of appropriate bound lengths. This requires that thorough bounds checking is performed on memory inspected from the guest and requires reasonable stopping points for data structures that need the size parameter to be inferred from guest memory. Setting limits also protects against maliciously linked

recursive data structures. This ensures that a compromised guest cannot affect probe behavior.

R2 requires that a malicious guest can neither modify nor prevent the logging of an event ϵ_T occurring at time t_x in any time t_{x+n} the (i.e.: future actions in the guests cannot alter previously logged operations). Event-based logging ensures that the executions of probed instructions in the guest are captured in the log right away. Preventing log modifications by guests ensures that events captured cannot be deleted or modified by guests even if the guest reformats media or terminates.

R3 allows services built on top of the log to decide how much trust to place in events captured in the log after a modification event. For instance, if an administrator observes a modification event ϵ_{mod} at time t_x she can decide to trust or not trust the events logged after time t_x depending on other available information. For example, a non-malicious kprobe may have caused the modification event. We only guarantee that the event will appear in the log and leave any event classification up to higher-level services. Recording potential attempts to circumvent logging ensures that higher-level applications have sufficient information to classify events. **R4** guarantees that an attacker cannot circumvent logging by simply redirecting calls to the functions of interest. In our case, that means write protecting both `system_call` and `sys_call_table` and the preceding call stack. The call stack includes the `sys_call_table` indicating the memory locations of the specific system call handlers, the general system call handler `system_call` and the hardware registers indicating which block of code to execute after performing an interrupt. In the case of hooking only the general system call handler, only modifications to `system_call` and the hardware registers (such as the `idt` register) must be monitored. We elaborate more on monitoring events and the specific registers that need to be monitored in Section 3.2.2.

When combined with **R2**, **R3**, and **R4**, **R5** can guarantee that every event of interest up to and including an attempt to circumvent logging is recorded ensuring completeness. Log completeness is required because the trust placed in the events occurring at some time point t_x relies upon the integrity of every event logged between when logging can begin, time t_0, and the time of the event immediately preceding event x at time t_{x-1}. If any event before event x is determined to be malicious, then we may decide that the details of an event occurring at time t_x cannot be trusted. Thus, if a service cannot review the

events occurring between time t_0 and when a probe is inserted, the service cannot determine the integrity of any event. The situation of missing data can occur during guest boot; exiting tools built on event based probing insert probes at some time t_{b+n}. If a malicious action occurs between when the system boots, t_b, and when the probe is inserted, then it will go unlogged by directly applying the instruction replacement event based probing technique mentioned in literature. We present a method to guarantee a probe is inserted before any invocation of the instruction it is replacing. To the best of our knowledge this paper is the first work that considers completeness of guest kernel-based events logged using trusted probes. We do not currently support the ability to log dynamically generated code that modifies itself after boot. Code generated as part of the boot sequence and never again can be probed successfully.

Apart from the requirements **R1** – **R5**, an additional goal of our system is to impose low overheads to remain practical. **D1** dictates that any attempts at logging must pose minimal performance impact on guests while also being transparent (**D3**). These two design goals ensure that any ensuing architecture remain feasible for production workloads. Our final design goal (**D2**) requires that we keep probing functions to the minimal required for logging in order to minimize additional attack surface. Below we discuss how we can achieve these goals through the use of probing techniques, novel guest boot sequence analysis, and well placed probes. The result is a secure and trustworthy logging service on which meaningful higher-level services can be built.

CHAPTER 2

BACKGROUND

2.1 Hardware Assisted Virtualization

The x86 architecture was not originally designed with virtualization in mind, but as VM's became popular hardware manufacturers looked at ways to improve their performance and robustness. Both AMD and Intel have released support for HAV in the form of extensions to the x86 instruction set.

HAV allows a VM to execute instructions natively on the hypervisor's CPU(s). However, the hypervisor must maintain control of the VM's execution. When the CPU is executing a VM's instructions, `VMExit` events are generated for any privileged operations that the VM attempts. A `VMExit` transfers control from the VM to the hypervisor allowing the hypervisor to perform any necessary operations before returning control back to the VM.

While allowing for robust and simplified hypervisor software, `VMExit`'s do incur performance overhead. Historically, one of the major causes of overhead in HAV was due to page faults in the VM. In earlier HAV implementations, every page fault would result in a `VMExit` since the guest could not control its own page tables. To alleviate this, vendors introduced a technique called two-dimensional page tables (TDP). In this paper we utilize Intel's TDP implementation, known as Extended Page Tables (EPT). The techniques apply to AMD's equivalent Nested Page Tables (NPT).

EPT allows VM's to manage their own page tables by managing guest-physical to host-physical address translations in hardware, effectively eliminating `VMExit`'s on page faults.

In EPT, there are two layers of page tables that must be traversed for guest memory accesses. Similar to conventional x86 page tables, EPT also provides a set of access flags that can be set at the page level: execute enable, write enable, and read enable. A `VMExit` is triggered on accesses that violate

the access flags due to an EPT violation. For example, if writes have been disabled for a page within a guest, an EPT violation `VMExit` is triggered on any write attempts to that page, and must be handled within the hypervisor. We later show how EPT access flags can be used to guarantee that probing systems do not miss events of interest occurring within the guest, fulfilling **R5**. For more information on EPT we refer the reader to Volume 3 of the Intel Software Development Manuals [15]; for AMD's equivalent NPT the reader can refer to Volume 2 of AMD's Programmer Manual [16].

2.2 Virtual Machine Monitor based Probing

Our work focuses on an audit log of guest-instructions by recording key events to add an extra layer of protection. Here, we highlight the mechanism used to enable such logging. Event based probing using debugging techniques has been proposed and applied in a number of different contexts [14, 17, 13, 18]. Lengyel et al. use event based probing for dynamic analysis of malware with the goal of remaining undetected during monitoring [14]. Estrada et. al. show the effectiveness of similar techniques for reliability and security monitoring [13, 19]. XenProbes uses the technique for profiling performance inside guests [17] and Spider uses it for stealthy debugging [18].

All of these approaches utilize HAV to invoke `VMExit`s upon execution of `int3` (`0xCC`) instructions in the guest. The key feature of event based probing is that an instruction within an untrusted environment can be replaced by an instruction (`int3` in this case) that causes a hardware enforced trap (i.e., a `VMExit`) to transfer control flow to a trusted environment. After guest inspection is done, the original instruction is executed within the guest and the breakpoint is re-inserted before guest execution resumes. Because probes cause a `VMExit`, which is an expensive operation, one must carefully design services built on such probes to reduce the number of exit events while also ensuring enough information is available to ensure meaningful services can be developed utilizing the logged data. We do not consider the event based approached used by LibVMI [20] as it invokes a `VMExit` on every single instruction in the target page for the logged event. Such an approach causes high overhead and is intractable due to our performance requirement **D1**. Our approach gives users the flexibility to determine the overhead paid based

on the level of protection deemed necessary for a given application.

Instruction replacement techniques for event based monitoring differ substantially from the event based monitoring used in libraries such as LibVMI [20]. LibVMI "inserts" probes by simply marking the page containing the probed instruction as non-executable within the two dimensional page table structures. Any time an instruction on that page is executed, an exception is raised and the hypervisor must ensure the offending address is the "probed" instruction before performing any action. We use the finer grained instruction-level probes discussed in the literature and referenced above. Additionally, we limit ourselves to only on event based systems. The research community has shown that timer based guest introspection (passive monitoring) can be easily circumvented by a malicious or compromised guest [21, 22].

2.3 The Semantic Gap

Any Virtual Machine Introspection (VMI) application must cross the "Semantic Gap" - the gap faced by developers of code running within the Virtual Machine Monitor (VMM) that must inspect guest memory with no knowledge of the kernel data structures or memory layout of the guest. Much research has been done in this area, and we point the reader to the overview done by Hebbal et al. for a more thorough discussion of the issue and many of the proposed solutions [23]. For this work, we assume that the address of the `sys_exec` and `sys_open` calls in Linux, along with the offset at which the Linux kernel `.text` addressing begins are provided (this memory mapping is well documented [24]). The latter is needed in order to identify the guest physical locations of the above functions (which are loaded into memory before paging is enabled in the guest). In Section 3.2 we discuss in more detail why this is necessary.

We favor the approach of querying `System.Map` for the location of relevant functions due to ease of access; this approach has shown to be successful in the literature for providing a low cost method for crossing the semantic gap [5, 17, 9]. We limit our discussion to Linux guests as the open source nature of Linux lends itself to easier distribution of VAs, the focus of our IDS, but a similar approach of querying the debug symbols for the Windows kernel has

also been met with success [14, 9].

We have intentionally made an effort to keep probed functions to a minimum and have limited ourselves to probing simple kernel functions as opposed to kernel data structures in order to limit the size of the trusted compute base (**D2**). As we show in Section 4.2, initial results indicate that many attacks can be detected with minimal probing, thus reducing not only the impact of the semantic gap issue, but the size of the trusted compute base as well. We also assume that the guest kernel is booted using an attestation technique. This allows probes to trust that the kernel being probed preserves the few properties necessary for probing, such as the address of the instruction to be probed.

2.4 Virtual Appliances

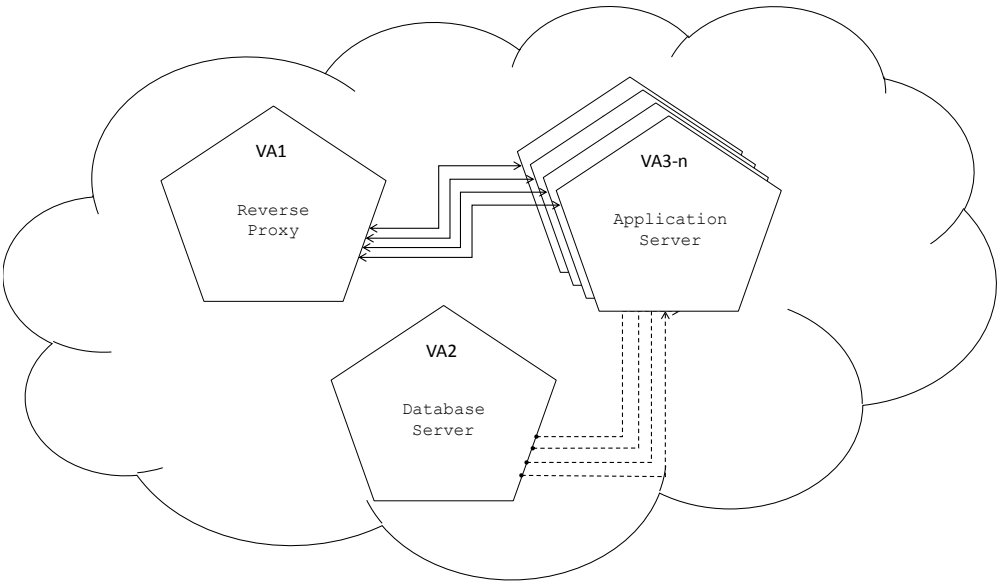

Figure 2.1: Typical Virtual Appliance Based Deployment of a Web Application

VAs are a popular method for deploying cloud services. One can simply choose an appliance from a list of images made available on a cloud provider's marketplace and immediately deploy services such as databases or web servers with minimal configuration. The tuned nature of these appliances makes their behavior more predicable than a VM used for general

purpose computation. In this paper we present an IDS that leverages the "appliance" nature of cloud based deployments instantiated using VAs. The IDS is built using guest event driven hypervisor-level probes to deliver relevant information to the policy compliance layer.

A typical deployment of a cloud based web application is shown in Figure 2.1 which shows a reverse proxy routing requests to an application processing layer, each of which communicate with a database before returning a response. We envision a system for which different policies protect each kind of VA. For Figure 2.1 there would be three main policies, one for VA1 the reverse proxy, one for VA2 the database and one for VA's 3 through N which serve as the application server(s). Policies can share layers if VAs are built using the same base distribution as a single distribution will have the same cron binaries running for example. While policies are stackable, the main advantages reside in the policies for each that differ, allowing for good coverage while limiting false positives. For example, a database server running MySQL should never execute a shell outside of configuration events; our monitoring system would detect such an operation as a violation. The event log can then also be used as compliance monitoring during configuration periods, and could serve as a method to detect insider threats attempting to re-configure applications in an attempt to cause unstable behavior.

CHAPTER 3

DESIGN

3.1 Attack Model

We assume that the hypervisor is a trusted entity and that the hypervisor side of the logging framework is secure. For the log file itself, a simple way to provide guarantees is to use remote logging, or approaches used in literature [25]. Here, we focus on the elements of logging that must be in place to facilitate proper logging of a guest that may become malicious at some point after boot. We assume that the hypervisor is using trusted boot, thus the integrity can be attested. Additionally, we assume that guests running on the hypervisor are also using attested boot mechanisms or guest kernels are known, non-malicious builds of Linux. This allows the hypervisor to guarantee the integrity of any guest kernel before the guest boots. We assume that the guest kernel is not malicious until after the first user-space program runs. This is a reasonable assumption as attempts to exploit a kernel will come from software loaded after boot (either malicious software will be loaded or vulnerable software exploited). Attacks can come in the form of modifications to guest memory, writes to guest registers (such as Model Specific Registers (MSRs), or the `IDT`) in an attempt to modify the location of the system call handler. We assume that the kernel can be fully compromised anytime *after* boot. Attacks can include loading kernel modules, modification of the kernel in place, or attempts to circumvent the scheduling of processes on the system. An attacker may try and copy the page of memory with the replaced instruction, fix said instruction, and redirect system calls to this new page. Such a redirect would either require modification of the Interrupt Descriptor Table in memory that is referenced to by the general system call handler or may come as a write to a hardware register in an effort to circumvent the code block executed after an interrupt.

3.2 Trustworthy Log

We start with our attack model and then show how through log acquisition and careful consideration of event types we can protect against such a model. Additionally, we explore the performance trade-off associated with log guarantees and highlight an approach we feel is a reasonable trade-off in providing a defense in depth solution. Based on our design requirements listed above, we guarantee that *certain* malicious activity within the guest can be logged and that *every* attempt to circumvent logging will be logged.

The guarantee we provide is that malicious activity with the goal of bypassing the logging mechanism will appear in the output log. These log events are the minimal required to enable trustworthy logging. We also show how relaxing this guarantee allows for much faster performance of the logging interface without a large increase in attack space that cannot be monitored. With a trustworthy logging mechanism in place, we can consider other events for logging that will reduce the size of the attack space that will go unlogged. The goal is to provide enough logging that an attacker will have difficulty in launching meaningful attacks while going unnoticed.

3.2.1 Log Acquisition

Figure 3.1 highlights how we probe the Linux kernel system calls `sys_exec` and `sys_open`. As mentioned above, we utilize an `int3` based probing mechanism to replace instructions in the guest kernel, ensuring information is logged anytime the affected functions are called, fulfilling **R2**.

To ensure that any attempt to modify a probe is logged (**R3**) we use EPTs to remove write permissions for the affected page, register a callback to handle these EPT violations, and within the callback handler only log attempts to modify the affected page if the violation occurs for the guest virtual address on which we inserted the probe. This gives administrators knowledge of attempts to subvert the logging system. While kprobes within the guest might cause non-malicious writes to locations of logging probes, an administrator would know the event is benign. Event classification is left to higher level services, we simply guarantee that modification events do appear in the log. Logging code also remains small, making formal verification more feasible. There are only 72 and 41 lines of code for our `sys_exec` logger and

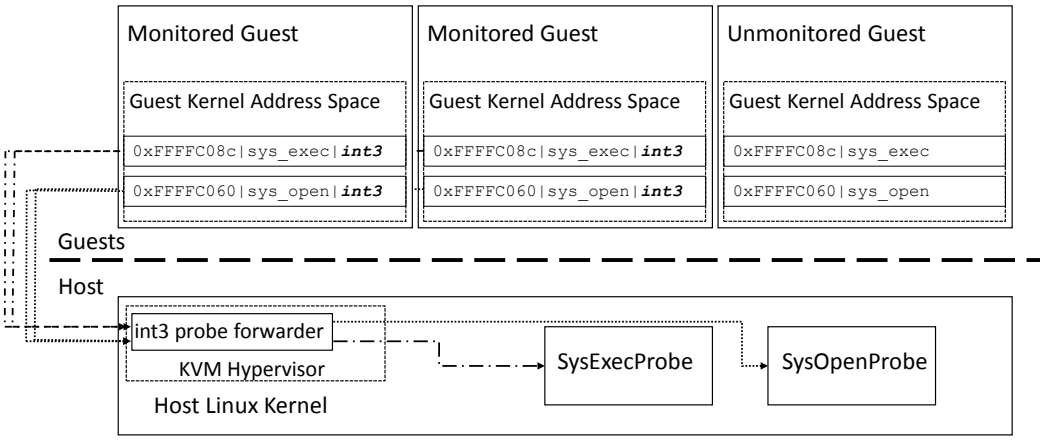

Figure 3.1: Event Driven Probe Architecture

`sys_open` logger respectively (not including the code required to insert the probes), keeping in line with **D2**. To ensure **R4** is met, we must consider every attack vector that could be used to circumvent logging. For every attack A_T there must be an event corresponding to actions of type T. Consider the following list of attacks that could circumvent logging:

A1 Rewrite replaced instruction(s) with the original instruction.

A2 Rewrite the general system call handler to reference a new, attacker supplied, Interrupt Descriptor Table.

A3 Rewrite the entry for the specific system call being hooked in the Interrupt Descriptor Table to point to an attacker supplied handle for the system call.

A4 Write to either the `IDTR` register (for legacy `int $80` based system calls) or various MSRs for so called "fast" system calls to force the hardware to invoke a malicious code block after interrupts (See Section 3.2.2 for a more detailed discussion of the specific registers).

A5 Probabilistically insert an interrupt after a system call (that is being logged) is made. Upon interruption, modify the `thread_struct` of the system call invoking process to point to a different system call handler upon being re-scheduled.

In section 3.2.2 we highlight how each attack is accounted for through hardware enforced events. It is worth noting here that **A5** would require

careful timing and is unclear if such an attack can be carried out successfully. Assuming it can be, such an attack would be probabilistic and not guaranteed to work for every system call made. We later discuss how removing the **A5** constraint greatly reduces the performance impact and we believe it has minimal affects on the overall trustworthyness of our logging architecture. In future work, we will look at dynamically paying the performance penalty to protect against **A5** by analyzing guests and dynamically moving probe locations if an attack is more probable based on logged data.

In order to ensure log completeness and fulfill **R5** we must place probes in their respective locations *before* the instructions at those locations are executed. The system calls being probed will be loaded at a predictable location within the guest physical memory (as noted in Linux's memory mapping documentation [24]). The knowledge of these locations allows us to determine the page number indicating the page containing the target instruction, which we use to watch for EPT violations of any guest physical address that occurs on the same page as an instruction of interest during the guest boot sequence. We are able to watch for such violations by utilizing a callback handler that gets called after KVM performs any necessary actions to handle the violation. Upon observing the first write violation for any address within the page of interest, we remove the execute bit from that page, allowing our callback handler to be invoked if any instruction on the page is executed. Subsequently, upon observation of any instruction execution on the page of interest, we know that the remaining code for that page must be loaded and can safely insert the probe. Having inserted the probe, we restore EPT permissions to allow execution and remove our checks for EPT violations due to execution exceptions on the page in which the probe is inserted as the checks are only required as the final step before probe insertion. By inserting probes in this manner during boot of guests, we are able to ensure log completeness and log every call to these two system calls, even while the first userspace applications are being started.

Finally, we must ensure that the actions taken within the probe do not place unwarranted trust in data obtained from the guest (**R1**). For example, our `sys_exec` logger logs two variable length string arrays. While these string are typically `\0` terminated, the guest could point the probe to a location with an arbitrarily large number of bytes before a `\0` is encountered. To protect against copying strings from guest memory, we only copy 500 bytes

and place a \0 at the 500th byte. While we may log garbage data in cases of an intentionally malicious guest and may truncate binary names in the case of exceptionally long, but legitimate, calls to `sys_exec`, this is a necessary trade off to ensure the probing interface remains resilient. Potential for truncating can be seen again when iterating through variable length arrays, which should be `NULL` terminated. We only iterate over up to 50 entries and exit iteration if `NULL` is encountered (in a legitimate case) and stop at 50 in the case of a malicious guest pointing the probe to a random memory location. Again, this has the side effect of potentially truncating logged arguments. In our experiments, we never truncated any legitimate data. Logging code also remains small, making formal verification more feasible. There are only 72 and 41 lines of code for our `sys_exec` logger and `sys_open` logger respectively (not including the code required to insert the probes). In our experiments these length decisions did not have an affect on the effectiveness of the IDS.

3.2.2 Events Logged

In addition to the information listed for each event type as defined below, all events also include the `hostname` of the KVM hypervisor on which the event occurs, a `timestamp` for the event, and the `vmid` (the qemu-kvm process id of the VM on the host on which the event is logged).

The three event types currently implemented in our system, and information collected unique to that type, are as follows:

- T_{se} - `sys_exec` events containing: `filename, argv,` and `envp`
  - `filename` - a \0 delineated string.
  - `argv` - a `NULL` delineated variable length array containing string pointers.
  - `envp` - a `NULL` delineated variable length array containing string pointers.
- T_{so} - `sys_open` events containing: `filename, mode` and `flags`.
  - `filename` - a \0 delineated string.
  - `flags` - an integer flags variable indicating options for the file.
  - `mode` - an integer indicating the mode for the file being opened.

- T_{mod} - Probe modification events containing: `gva`
  - `gva` - A long integer indicating the guest virtual address being modified.

The following two events are unique to logging system calls and provide guarantees that malware within the guest is unable to circumvent the logging mechanism. These require additional callbacks be provided by the underlying probing framework; we save implementation of these events for future work. These two events are not currently provided by any event based monitoring framework in the literature [13, 17, 14].

- T_{lidt} - `lidt` event. Triggered on execution of the `lidt` (Load interrupt descriptor table) x86 instruction.
- T_{wrmsr} - `wrmsr` event. Triggered on execution of the `wrmsr` (Write Model Specific Regsiter) x86 instruction.

These two events are hardware enforced; once the hypervisor has configured the processor to trap these calls, their execution will always force a `VMExit`. The `lidt` trap can be configured by setting bit 2 (Descriptor Table Exiting) of the `MSR_IA32_VMX_PROCBASED_CTLS2` model specific register to 1 within the hypervisor before VM's are started. Similarly, writes to model specific registers within the guests can be trapped by setting bit 28 of the same model specific register to 0. For `int $80` based system calls, the `lidt` trap is sufficient. For `sysenter` invoked system calls, the three MSRs `IA32_SYSENTER_{ES, EIP, ESP}` must be monitored through the `wrmsr` trap. Finally, for `syscall` invoked system calls, the MSR `IA32_LSTAR` must be monitored with the `wrmsr` trap. The registers listed above are used to register Interrupt Service Routines (ISRs) with the processor; in Linux, these point to the general system call handler. The performance impact of these two events is negligible under normal operation as these events occur only during boot of the guest kernel and during configuration of MSRs.

Let us now consider how these event types can protect against the attacks listed above. Attacks **A1**, **A2**, and **A3** can be protected by properly removing the write enable bits for the pages containing the instruction modified, the general system call handler, and the interrupt descriptor table and listening to events of type T_{mod}. The event T_{mod} is hardware enforced by EPT.

Attempts to modify pages for which the write enable bit has been removed will trigger a `VMExit` through an EPT violation. Attacks that try to change the ISR for system calls (**A4** above) can be logged with events of type T_{lidt} and T_{wrmsr}; again, these are hardware enforced events. Finally, careful placement of probes can ensure that logging occurs before interrupts have been re-enabled by placing the probe on the general system call handler, mitigating attack **A5**. Mitigating **A5** does have high performance impact as we discuss in Section 3.3; we believe placing the probe at the general system call handler is a reasonable trade-off as attacks of this kind would be unreliable.

Note that many more event types are possible as event based probing provides a trusted mechanism with which to hook any kernel function. But in keeping with **D1** and **D2**, we choose to keep this number small. In this work, each event type T above corresponds to an equivalent probe P_T inserted into each guest. An interesting area for future research (discussed more in Section 6) is building new event types based on the output of the above events types. For instance, a new `apt-get` type could be defined to allow an administrator to write fine grained policies regarding arguments to `apt-get` on a particular machine. These `apt-get` events would be not require additional probing as all the information required is already stored within the `sys_exec` event. Such an approach would allow for more dynamic auditing approaches while not impacting the size of the trusted compute base as any such event types would be based on the trusted log and not on the addition of (potentially fragile) probes into guest memory. Future work will explore adding more probes in to improve the efficacy of higher level services while minimizing performance costs to guests.

3.2.3 Logging Format

At this point, we are placing all probe output into `/var/log/kern.log` and then processing the output with a user space application to build and then processes events. This process is show in Figure. 3.2 below. In order to allow for easier processing by higher level applications, we adhere to a JSON like format when doing logging within the host kernel.

A log sample for a `touch text.log` event looks like the following:

Listing 3.1: Example `sys_open` Probe Output

```
{"VMID": 1884, "LOGGER": "SYS_OPEN_LOGGER", "KIND": "BEGIN"}
{"VMID": 1884, "LOGGER": "SYS_OPEN_LOGGER", "KIND": "ARG",
 "ARG_NAME": "filename", "VALUE": "test.text"}
{"VMID": 1884, "LOGGER": "SYS_OPEN_LOGGER", "KIND": "ARG",
 "ARG_NAME": "flags", "VALUE": "0x941"}
{"VMID": 1884, "LOGGER": "SYS_OPEN_LOGGER", "KIND": "ARG",
 "ARG_NAME": "mode", "VALUE": "0x1b6"}
{"VMID": 1884, "LOGGER": "SYS_OPEN_LOGGER", "KIND": "END"}
```

The `TIMESTAMP, HOSTNAME` and `LOG_ID` are also included and are set by the `printk` function within the hypervisor. We trust these fields to be accurate when read by higher level tools. The accuracy of these fields is important as will be discussed in the next section on the development of higher level tools.

For each event, we have a `BEGIN` statement and an `END` statement. Everything in between those statements make up the body of the event and are used to log parameters read from the guest.

3.3 Intrusion Detection System for Virtual Appliances

To highlight our approach to services built on top of an event based trustworthy log, we have developed an IDS which triggers alerts on violations of filename white-lists. The checks are performed on filenames passed to guest `sys_exec` and `sys_open` calls. To enable ease of use, we have also built a policy recorder that translates guest events to white-list policies during recording.

The architecture of our intrusion detection system is show in Figure 3.2. Raw probe logs are transferred from kernel to user space using the `/var/log/kern.log` interface. From there, the logs are placed in a buffer as they are read from the file. An `ioctl` interface to `/var/log/kern.log` is used to ensure updates are pushed to the user space application as soon as probes write to the file. Within the user space event parser, buffers must be used to ensure that output from a probe P^1_{so} into guest G_1 do not become integrated into an event ϵ_2 from the output of the probe P^2_{so} placed into guest G_2, as the arrival of such logs may be intermingled within `/var/log/kern.log`. This is ensured by placing all logs from a given probe into a unique buffer identified by the `LOGGER_TYPE,HOSTNAME,VMID` sequence. As mentioned previously, since

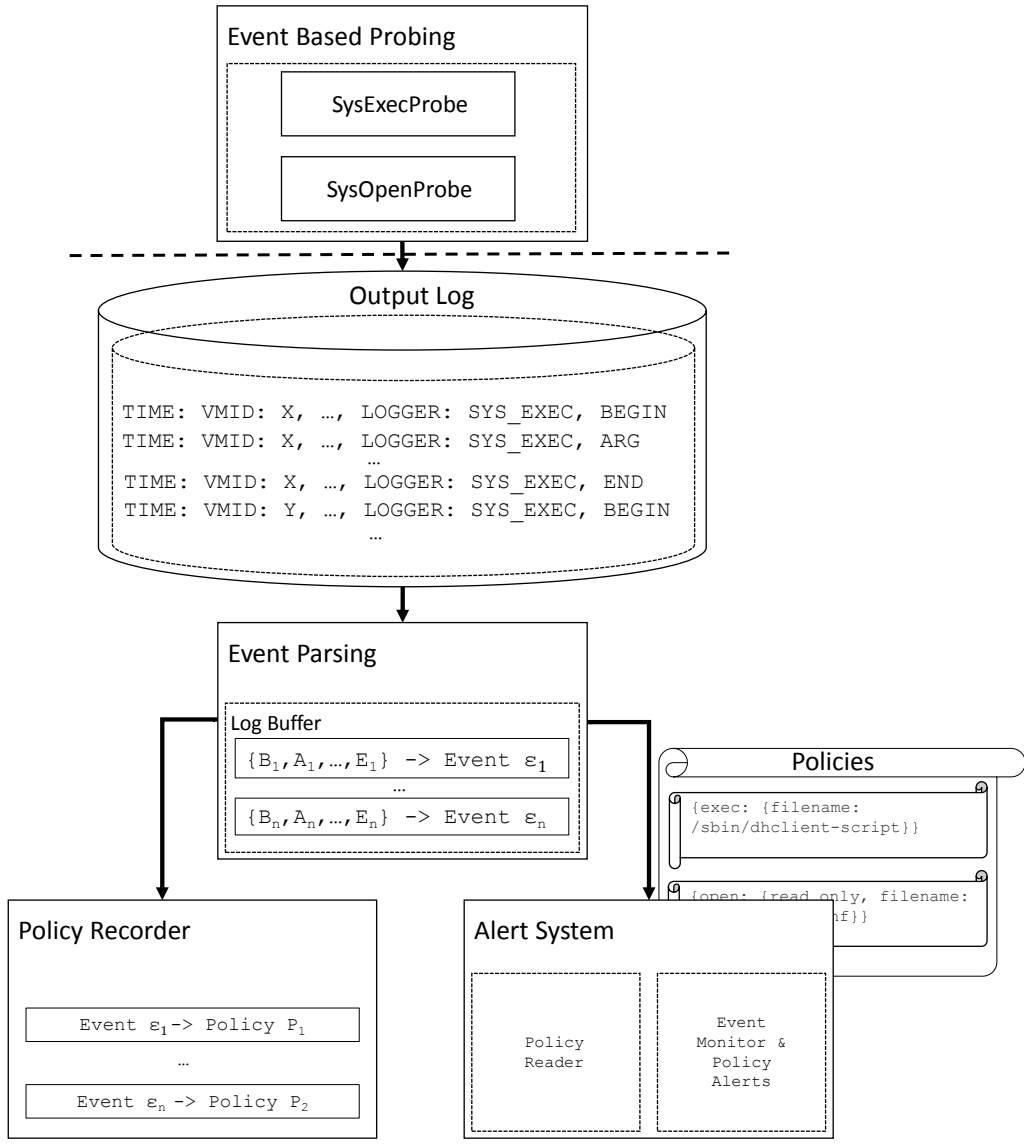

Figure 3.2: Trustworthy-Log Driven IDS Architecture

the buffer being used is determined by these sequence of values, these values must be set by the hypervisor. To ensure the guest can not impact actions taken by the logging system, no value read from the guest is used to identify a probed event or which buffer in which to place a logged statement. For now, we do not consider multiple vCPU guests, thus only need to worry about intermingling between guests. In the case of multiple vCPUs, the vCPU id would also need to be used as a unique identifier as it would be possible that a probed location be called from multiple vCPUs simultaneously. Extending this approach to multiple vCPU based guests will be done in future work.

We note here however that there are certain limitations to our approach that would allow an attacker to commit a malicious action without being logged. Consider a vulnerable binary running on a system that is compromised through a buffer overflow attack. Assuming the attacker does not crash the binary, it might be possible to run code under the guise of an already executing process. As long as the payload never opened a file or executed another binary, it would go unlogged. However, our approach substantially reduces the actions that can be taken by an attacker. Adding a separate event for system calls dealing with network access further mitigates the possibility that a malicious payload is able to do any useful work without being logged. This can be combined with ASLR, non-executable heaps and other defenses to increase the cost of a successful attack. On the other hand, all attempts to modify the logging facility are logged (depending on the performance trade-off chosen for a given guest).

After event parsing is complete, processed events are passed to either a policy recording layer or an alert system for our IDS. The policy recording system allows an administrator to record standard behavior for a VA in terms of white-listing the actions taken during policy recording. Listing 3.2 shows an example policy built using our policy recorder while executing the `which` command on a guest under inspection. Currently, white-lists are separated from attackers executing in the guest by the VMM. In future work we will investigate using attestation mechanisms for the white-list and while-list enforcing mechanism.

Listing 3.2: Example `which.policy` file

```
{ "policies": [
{"exec": {"type": "whitelist","filename":"/usr/bin/which"}},
{"open": {"type": "whitelist","access_type": "read",
  "filename": "/etc/ld.so.cache"}},
{"open": {"type": "whitelist","access_type": "read",
  "filename": "/lib/x86_64-linux-gnu/libc.so.6"}},
{"open": {"type": "whitelist","access_type": "read",
  "filename": "/usr/bin/which"}}
]}
```

CHAPTER 4

EVALUATION

In this section we evaluate both the impact of the probes on the performance of the guest and on the ability of the IDS to detect a real world attack on a popular cloud based web application.

4.1 Performance

To evaluate the overhead of our probing mechanisms driven by guest events, we run three benchmarks that are representative of cloud workloads. These include:

- Apache Bench - a web serving benchmark for the Apache web server, [26],
- Redis Bench - a benchmark for the in memory data store [27],
- OpenSSL Profiling - used to understand the impact on encrypted communication within guests.

These tests were chosen because they represent a disk-read heavy workload (Apache), network heavy workload (Redis, Apache), and a CPU heavy workload (OpenSSL). Web applications will often call in memory caches before sending a response using Apache configured with OpenSSL. All tests are configured using the Phoronix Test Suite and are run 90 times each. The first 30 runs are performed with our trusted probes loaded and then we run 30 without. The last 30 runs are done while having probes loaded at the general system call handler, before interrupts have been re-enabled in the guest to highlight the performance penalty paid while protecting against **A5**. Figure. 4.1 shows the results for both Apache Bench and for OpenSSL. The results have been normalized to running in a guest without probing. Apache bench

results are in terms of requests served per second and those for OpenSSL are in terms of signatures generated per second, but here both have simply been normalized to highlight the percentage decrease in performance caused by probing. Figure. 4.2 is for the Redis benchmark, which runs five separate requests types to the in memory data store.

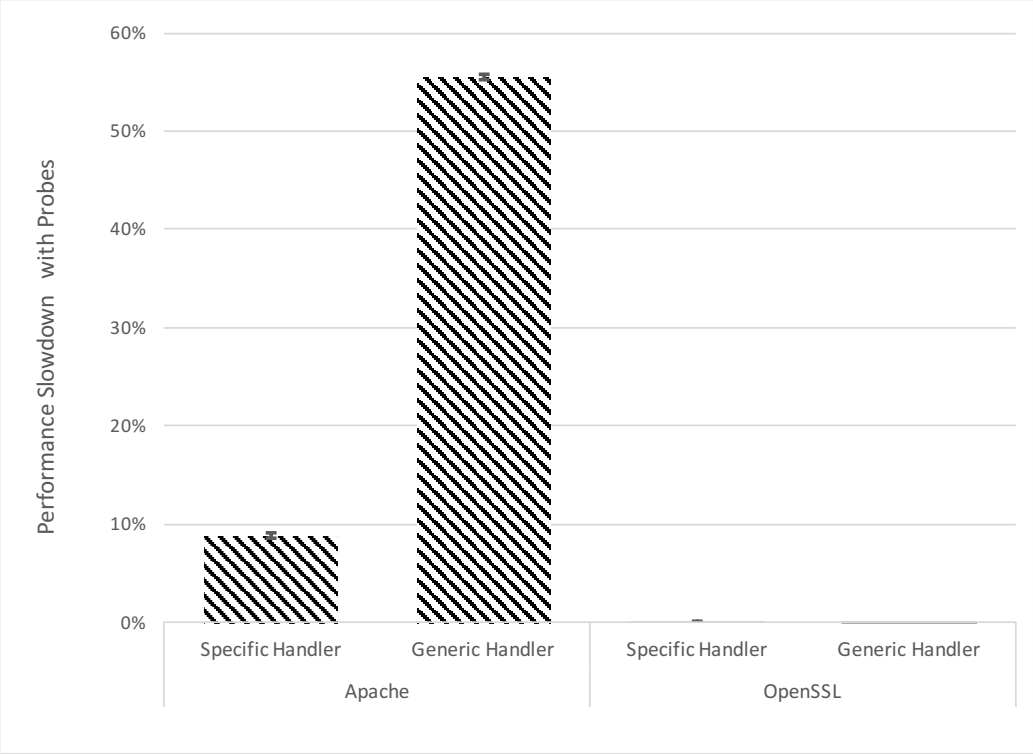

Figure 4.1: Apache Bench and OpenSSL Overhead Relative to Running with no Probing.

In the case of hooking specific system call handlers, it is clear to see that overheads remain tolerable (less than 10%), because we are only probing two guest kernel functions. The overheads are large when protecting against **A5** though, around 55% for Apache and 75% for Redis. Notably, we see very little slow down for OpenSSL in both cases. This is because OpenSSL does not have to interact with the kernel as much as Apache and Redis to complete its workload. OpenSSL works by loading a key in memory and then generating signatures using that key. It is up to another process, Apache in our case, to write out any information to the network. Apache and Redis are both opening sockets and sending data over the network, which is why we see a much higher penalty being paid when hooking the generic system call

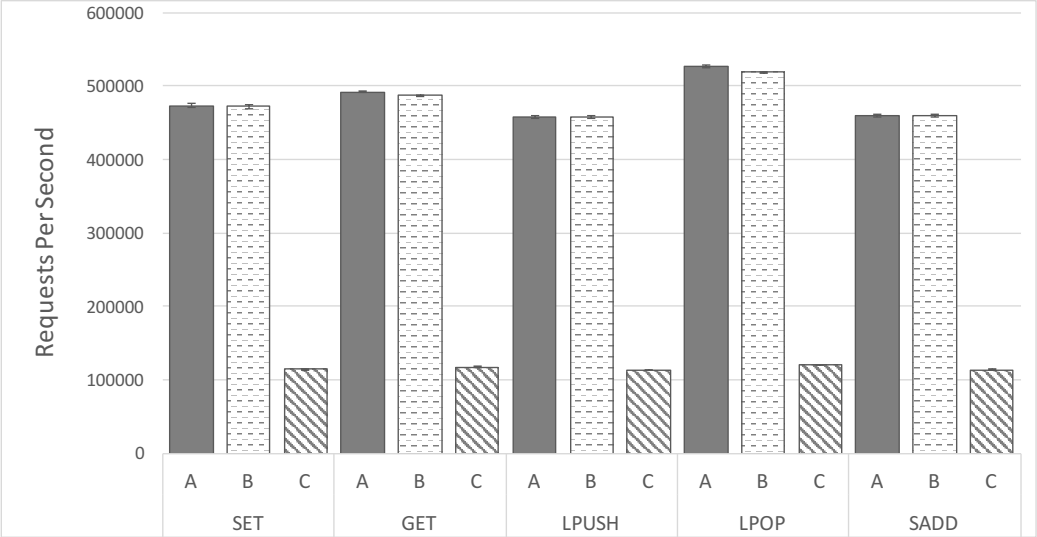

Figure 4.2: Redis Benchmark Overhead for 5 Redis Operations.

(A) is without probing the guest, (B) is probing only the specific system call handlers, and (C) is probing the general system call handler.

handler. We feel that the protections against **A1**, **A2**, **A3**, **A4** (requirements **D3** and **R4**) go a long way in protecting the specific system call handler, substantially reducing the unloggable attack space when hooking only the specific system call handlers. In future work, we will investigate dynamically choosing probe location based on observed events. Such an approach could utilize game theory to model situations in which inserting probes at the general system call handler is worth the performance penalty. We found that the majority of overhead when hooking the specific system call handlers comes from the `sys_open` probe due to the large number of times that system call is used.

4.2 IDS Evaluation

We evaluate the efficacy of the IDS built on top of our trusted logging platform by looking at real world exploits for motivation. In a recent attack on the website for the Linux distribution Linux Mint [28], attackers were able to gain shell access as the `www-data` user, the user typically reserved for only running the `httpd` process [29]. The attack exploited a vulnerability in the popular blogging framework, Wordpress. Wordpress is representative

of a typical cloud application as it can be deployed on many VAs to enable horizontal scalability as show in Figure. 2.1. To see how our system would have handled such an attack, we installed a copy of Wordpress and a typical plugin and attacked the setup using Wordpress Vulnerability Database ID #8209 [30].

We first setup a Wordpress application server and separate database server to act as our VAs. Since our IDS supports policy stacking, we are able to record a separate policy for Wordpress and use the `dhcp.policy` file common to all VAs built using the same base Ubuntu 14.04 LTS distribution. Including that policy is necessary as it removes the chance of false positives every time a dhcp lease renewal is performed. It would not be necessary for VAs using static IP's. An abridged version of the wordpress policy file is show in Listing 4.1. Our policy recording utility produced a policy that served as a starting point and then we used knowledge about proper wordpress installs to fine tune the policy. For example, the policy recording utility produced many single `filename: /var/www/html/*.php` entries. We removed these and converted it into a single `directory: /var/www/html` entry as shown on the first line of the policy in the listing.

Listing 4.1: Abridged `wordpress.policy` file

```
{"open": {"type": "whitelist","access_type": "read",
  "directory": "/var/www/html"}},

{"open": {"type": "whitelist","access_type":"create",
  "directory": "/var/www/html/wp-content/uploads"}},
{"open": {"type": "whitelist","access_type":"modification",
  "directory": "/var/www/html/wp-content/uploads"}},
{"open": {"type": "whitelist","access_type":"read",
  "directory": "/var/www/html/wp-content/uploads"}},

{"open": {"type": "whitelist","access_type":"create",
  "directory": "/var/www/html/wp-content/plugins"}},
{"open": {"type": "whitelist","access_type":"modification",
  "directory": "/var/www/html/wp-content/plugins"}},
{"open": {"type": "whitelist","access_type":"read",
  "directory": "/var/www/html/wp-content/plugins"}},
```

```
{"open": {"type": "whitelist","access_type":"create",
  "directory": "/var/www/html/wp-content"}},
{"open": {"type": "whitelist","access_type":"modification",
  "directory": "/var/www/html/wp-content"}},
{"open": {"type": "whitelist","access_type":"read",
  "directory": "/var/www/html/wp-content"}},
```

We exploit the vulnerability using Metasploit [31] to determine if our alerting system is able to capture anomalous events. Because the exploit works by injecting arbitrary PHP code, we can only detect attacks that use PHP to access other files on the system (outside of the /var/www/html directory) or execute system binaries. We detect the exploit immediately upon the attack dropping into a shell, as `/bin/sh` should never execute on the system. We could detect the exploit sooner by adding an extra probe to `sys_socket`. In future work, we will explore detection coverage and delay in relationship to the overheads paid by guests (when only hooking specific system call handlers) to determine which functions to probe.

Our approach relies on the fact that many exploits require a binary to load and execute on a system. And if the exploit does not run in a separate process, as is the case in the example given above, the attacker will likely either execute a system binary or open a file, revealing malicious activity (assuming an oracle exists that can classify logged events). For instance, the loading of kernel modules could be audited by looking at events of type T_{se} with `filename` equal to `insmod`. This would potentially reveal the loading of a rootkit. We note here however that there are certain limitations to our approach that would allow an attacker to commit a malicious action without being logged. Consider a vulnerable binary running on a system that is compromised through a buffer overflow attack. Assuming the attacker does not crash the binary, it might be possible to run code under an already executing binary. Payload code could then explore the full system call interface and potentially exploit the running kernel. Such an event would not be logged, though any attempt to remove our probe using such an exploit would be noted in the log. While the attack event itself would not be logged, any rootkit loaded in such a manner could not hide invocations of a userspace application making `sys_exec` and `sys_open` system calls. This increases the

burden of carrying out a successful attacks as malicious payloads will have to be carried out within a vulnerable binary or the kernel to go undetected. In future work we will explore creating probes for the most vulnerable locations within the Linux kernel by evaluating past exploits.

CHAPTER 5

RELATED WORK

Huh et al. discuss a trusted logging architecture for grid computing using Xen [25]. Their approach relies on logging events as they are intercepted by Xen device drivers. Our trusted logging is more flexible as any action within the guest can be logged on instruction execution. Additionally, the authors propose an extensive architecture for guaranteeing the log is not fabricated by the provider. We view this work as complementary. Thus far, we have focused on trust related issues related to log generation and can utilize similar techniques for improving trustworthiness.

Montanari et al. discuss using VMI for integrity checking of credit card security policies for monitored guests [32]. We share similar goals in that the authors wanted to perform compliance auditing with the least amount of evidence. Our system builds on or extends on these ideas by implementing a trusted event based logging system on which compliance audits could then be performed.

Crawford et al. discuss a methodology for detecting insider threats that relies on scanning the memory of running virtual machines every 30 minutes [11]. As we discussed earlier, polling techniques such as this are limited in that they are easily circumvented, giving attackers a 30 minute window in which to perform malicious activities. Kienzle et al. explore using VMI techniques for endpoint configuration compliance, but require the compliance audit package run in a separate VM, increasing the resources of the monitor [33]. Their approach to compliance also relies on polling, thus can be circumvented. Our approach provides a trusted log which is guaranteed to capture every event probed. In future work, we intend to use our trusted event log to perform compliance checks of Mandatory Access Control (MAC) systems running within the guests. Win et al. propose using VMI to provide additional layers of security for a similar system, but rely on information from a trusted in-guest monitoring agent to report relevant accesses to a

trusted compliance layer VM [34]. Our approach places no trust in the guest after the initial kernel is loaded using an attestation technique provided by a TPM.

KvmSec is a security extension for KVM, but relies on probes running in untrusted guests [35]. Numerous papers have been published regarding detection of specific kinds of malware. For example, Liu et al. address issues related to the "Heartbleed" OpenSSL vulnerability using a VMM [12]. AntFarm and Lycosid both present ways to address the semantic gap and track running processes on guests [7, 2]. Our approach uses event based probing, thus will not miss events while having less overhead when compared to a system that requires running a separate trusted VM from which to perform monitoring.

In "Space Traveling across VM" [8], the authors cross the semantic gap by relying on an additional virtual machine from which to run probes. This approach has a large over head, thus would violate **R6**. Techniques like "Virtuoso" are complimentary to our trusted log and could be used to inform future probes of relevant locations within the guest for probing [36]. With regards to work related to IDS, Kosoresow and Hofmeyr show the effectiveness of system call traces by using temporal patterns of system calls to detect intrusions [37]. While the IDS presented here relies upon white-listing, their technique could also be applied. Performance considerations and techniques to improve the performance of logging with kprobes is discussed by Feng et al. [38]. While our approach does not use kprobes, their techniques for improving the performance of the communication between kernel space and user space may prove to be useful in future work.

CHAPTER 6

CONCLUSION & FUTURE WORK

In this paper we have shown the events that must be logged when probing guest instructions from within a VMM to ensure attempts to circumvent logging can be audited by higher level services. We show how existing instruction replacement based probing mechanisms must be extended to include a mechanism to guarantee that every invocation of a probed instruction triggers an event. We do this by inducing a unique sequence of EPT violations to guarantee probes placed in a guest kernel are placed before the instruction can be invoked. We also identify two new events that must be added to ensure attempts to circumvent logging can be audited. These events will allow writes to MSRs and the `idt` to be audited to determine if an attacker is attempting to circumvent probes placed in system calls.

We highlight a methodology for creating trustworthy services built on top of instruction based probes. Namely, attempts to circumvent probes must be well understood and handled appropriately. This requires one consider not just protection of the instruction being replaced by the probing mechanism, but protection for every instruction that transfers control flow to the probed instruction. We highlight five requirements that drive this methodology. These include control flow considerations (**R1**), log integrity protections (**R2**), probe robustness (**R3**), calling function considerations (**R4**), and a requirement that logged data contain every event up to and including an attempt to circumvent logging (**R5**). Additionally, we try to adhere to design goals to minimize the performance impact on guests (**D1**), reduce additions to the trusted compute base (**D2**) and require not guest modifications (**D3**). We also outline potential attacks against an event-driven log and show how such attacks can be audited.

To highlight how higher level services can be built on a trusted log, we developed a white-list based IDS and policy recording system. The IDS aims to provide defense-in-depth to VAs. Cloud computing and the prevalence

of VAs allows for more complete white-listing with fewer false positives as VAs are commonly deployed to perform a single task. Any deviation from that task is a potential security violation. We demonstrate the effectiveness of our system using a real world vulnerability. Finally, we have shown the performance trade-offs required to protect against attack vectors when using hypervisor-based probing mechanisms. In particular, protection against the preemption attack vector requires a large overhead (55-75% in some benchmarks). Such an attack would be probabilistic and relaxing our logging system to not defend against this attack still greatly increases the cost of carrying out a successful attack against the logging system while providing more tolerable performance overhead (around 10% in the worst case among evaluated benchmarks).

In future work we will explore reducing the performance impact so that guests do not have to choose between protection guarantees and performance through the usage of Intel's new `#VE` exceptions that allow certain EPT violations to be handled in the guest. When combined with additional logging mechanisms in the hypervisor (to monitor register writes) this could allow operations causing the highest performance impact be moved to the guest. Any code base handling the exception could be protected using a unique EPT view (also controllable by guests) to ensure that a compromised kernel could not manipulate the exception handler.

REFERENCES

[1] [Online]. Available: https://aws.amazon.com/marketplace/

[2] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, *VMM-based hidden process detection and identification using Lycosid.* ACM, Mar. 2008.

[3] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Antfarm: Tracking processes in a virtual machine environment," in *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, 2006.

[4] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An Architecture for Secure Active Monitoring Using Virtualization," *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pp. 233–247, 2008.

[5] X. Jiang, X. Wang, and D. Xu, *Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction.* ACM, Oct. 2007.

[6] J. Lamps, I. Palmer, and R. Sprabery, "WinWizard: Expanding Xen with a LibVMI Intrusion Detection Tool." IEEE, 2014, pp. 849–856.

[7] S. T. Jones and A. C. Arpaci-Dusseau, "Antfarm: Tracking Processes in a Virtual Machine Environment." ..., 2006.

[8] Y. Fu and Z. Lin, "Space Traveling across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection," in *Symposium on Security and Privacy.* IEEE, 2012.

[9] A. S. Ibrahim, J. Hamlyn-Harris, J. Grundy, and M. Almorsy, "Cloudsec: A security monitoring appliance for virtual machines in the iaas cloud model," in *Network and System Security (NSS), 5th International Conference on*, 2011, pp. 113–120.

[10] J. Lamps, I. Palmer, and R. Sprabery, "WinWizard: Expanding Xen with a LibVMI Intrusion Detection Tool," in *Int. Conference on Cloud Computing (CLOUD '14).*

[11] M. Crawford and G. Peterson, "Insider Threat Detection using Virtual Machine Introspection," in *Hawaii International Conference on System Sciences (HICSS '13)*.

[12] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia, "Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation," in *22nd ACM Conference on Computer and Communications Security (CCS '15)*.

[13] Z. J. Estrada, C. Pham, F. Deng, L. Yan, Z. Kalbarczyk, and R. K. Iyer, "Dynamic vm dependability monitoring using hypervisor probes," in *Dependable Computing Conference (EDCC), 2015 Eleventh European*. IEEE, 2015, pp. 61–72.

[14] T. K. Lengyel, S. Maresca, B. D. Payne, G. D. Webster, S. Vogl, and A. Kiayias, "Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system," in *30th Annual Computer Security Applications Conference on (ACSAC)*, 2014.

[15] [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf

[16] [Online]. Available: http://support.amd.com/TechDocs/24593.pdf

[17] N. A. Quynh and K. Suzaki, "Xenprobes, a lightweight user-space probing framework for xen virtual machine," in *USENIX Annual Technical Conference (ATC '07)*.

[18] Z. Deng, X. Zhang, and D. Xu, "Spider: Stealthy binary program instrumentation and debugging via hardware virtualization," in *Proceedings of the 29th Annual Computer Security Applications Conference*. ACM, 2013, pp. 289–298.

[19] C. Pham, Z. Estrada, P. Cao, Z. Kalbarczyk, and R. K. Iyer, "Reliability and security monitoring of virtual machines using hardware architectural invariants," in *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*. IEEE, 2014, pp. 13–24.

[20] B. D. Payne, M. D. P. de Carbone, and W. Lee, "Secure and Flexible Monitoring of Virtual Machines," in *Annual Computer Security Applications Conf. (ACSAC '07)*.

[21] S. Bahram, X. Jiang, Z. Wang, M. Grace, J. Li, D. Srinivasan, J. Rhee, and D. Xu, "DKSM: Subverting Virtual Machine Introspection for Fun and Profit." IEEE, Oct. 2010, pp. 82–91.

[22] G. Wang, Z. J. Estrada, C. Pham, Z. Kalbarczyk, and R. K. Iyer, "Hypervisor introspection: A technique for evading passive virtual machine monitoring," in *9th USENIX Workshop on Offensive Technologies (WOOT 15)*. Washington, D.C.: USENIX Association, Aug. 2015. [Online]. Available: https://www.usenix.org/conference/woot15/workshop-program/presentation/wang

[23] Y. Hebbal, S. Laniepce, and J.-M. Menaud, "Virtual Machine Introspection: Techniques and Applications," *2015 10th International Conference on Availability, Reliability and Security (ARES)*, pp. 676–685, 2015.

[24] [Online]. Available: https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt

[25] J. H. Huh and A. Martin, "Trusted logging for grid computing," in *Third Asia-Pacific Trusted Infrastructure Technologies Conference (APTC '08).*, 2008.

[26] [Online]. Available: https://httpd.apache.org/docs/2.4/programs/ab.html

[27] [Online]. Available: http://redis.io/topics/benchmarks

[28] [Online]. Available: https://www.linuxmint.com/

[29] [Online]. Available: http://thehackernews.com/2016/02/linux-mint-hack.html

[30] [Online]. Available: https://wpvulndb.com/vulnerabilities/8209

[31] [Online]. Available: https://www.rapid7.com/db/modules/exploit/unix/webapp/wp_ajax_load_more_file_upload

[32] M. Montanari, J. H. Huh, D. Dagit, R. B. Bobba, and R. H. Campbell, "Evidence of log integrity in policy-based security monitoring," in *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012)*.

[33] D. Kienzle, R. Persaud, and M. Elder, "Endpoint Configuration Compliance Monitoring via Virtual Machine Introspection," *System Sciences (HICSS)*, 2010.

[34] T. Y. Win, H. Tianfield, and Q. Mair, "Virtualization security combining mandatory access control and virtual machine introspection," *IEEE/ACM 7th International Conference on Utility and Cloud Computing (UCC '14)*.

[35] R. D. P. Flavio Lombardi, "KvmSec: A Security Extension for Linux Kernel Virtual Machines," pp. 1–6, Oct. 2008.

[36] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection," in *Symposium on Security and Privacy*. IEEE, 2011.

[37] A. P. Kosoresow and S. A. Hofmeyer, "Intrusion detection via system call traces," *IEEE software*, vol. 14, no. 5, pp. 35–42, Jan. 1997.

[38] W. Feng, V. Vishwanath, and J. Leigh, "High-fidelity monitoring in virtual computing environments," 2007.

[39] M. Montanari and R. H. Campbell, "Attack-resilient compliance monitoring for large distributed infrastructure systems," in *5th International Conference on Network and System Security (NSS '11)*.

[40] M. Montanari and R. H. Campbell, "Confidentiality of event data in policy-based monitoring," in *Int. Conference on Dependable Systems and Networks (DSN '12)*.

[41] M. Zhang, D. Marino, and P. Efstathopoulos, "Harbormaster: Policy enforcement for containers," in *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, Nov 2015, pp. 355–362.

[42] J. H. Huh, M. Montanari, D. Dagit, R. B. Bobba, D. W. Kim, Y. Choi, and R. Campbell, "An empirical study on the software integrity of virtual appliances: are you really getting what you paid for?" in *8th ACM Symposium on Information, Computer and Communications Security (ASIACCS '13)*.