Stealthy Deployment and Execution of In-Guest Kernel Agents

Tzi-cker Chiueh   Matthew Conover   Maohua Lu   Bruce Montague

Symantec Research Labs

{tzi-cker_chiueh, matthew_conover, maohua_lu, bruce_montague}@symantec.com

Abstract

As more and more virtual machines (VM) are packed into a physical machine, refactoring common kernel components shared by virtual machines running on the same physical machine could significantly reduce the overall resource consumption. The refactored kernel component typically runs on a special VM called a virtual appliance. Because of the semantics gap in Hardware Abstraction Layer (HAL)-based virtualization, a physical machine's virtual appliance requires the support of per-VM in-guest agents to perform VM-specific operations such as kernel data structure access and modification. To simplify deployment, these agents must be injected into guest virtual machines without requiring any manual installation. Moreover, it is essential to protect the integrity of in-guest agents at run time, especially when the underlying refactored kernel service is security-related. This paper describes the design, implementation and evaluation of a stealthy agent deployment and execution mechanism called SADE that requires zero installation effort and effectively hides the execution of agent code. To demonstrate the efficacy of SADE, we describe a kernel memory scanning virtual appliance that uses SADE to inject its in-guest agents, and report SADE's start-up overhead and run-time performance penalty on this virtual appliance.

1 Introduction

One of the most compelling use cases of virtualization technology is its ability to consolidate applications originally running in multiple machines into a single physical server by executing each of them in a separate virtual machine (VM). For VMs running on the same physical machine, significant redundancy among them is likely to exist. Memory de-duplication, which is supported in VMware's ESX and some versions of KVM, aims to eliminate the memory resource waste due to such redundancy. However, because of mismatch in page alignment and use of physical pointers, not all memory-level redundancy can be effectively and efficiently uncovered. Another way to eliminate this inter-VM redundancy is through *kernel component refactoring*, which takes out common kernel components shared by VMs running on the same physical machine and runs them on a separate VM called a *virtual appliance*. In this virtual appliance execution architecture, the virtual appliance VM typically performs additional analysis, filtering or transformation operations on data traversing on the critical path of network, disk I/O or virtual memory accesses made by other user virtual machines. Example functionalities that a virtual appliance VM performs on behalf of user VMs include firewalling, anti-virus (AV) scanning, volume management, I/O change tracking, memory decompression and decryption, etc. This virtual appliance execution architecture entails several advantages. First, the overall resource usage is reduced because the inter-VM redundancy is minimized. Second, the development effort for common kernel components is also reduced because they now can be developed in a way that is largely independent of the platforms of the user VMs. Finally, for security-related kernel components, moving them into a separate VM renders them immune from kernel compromises in user VMs.

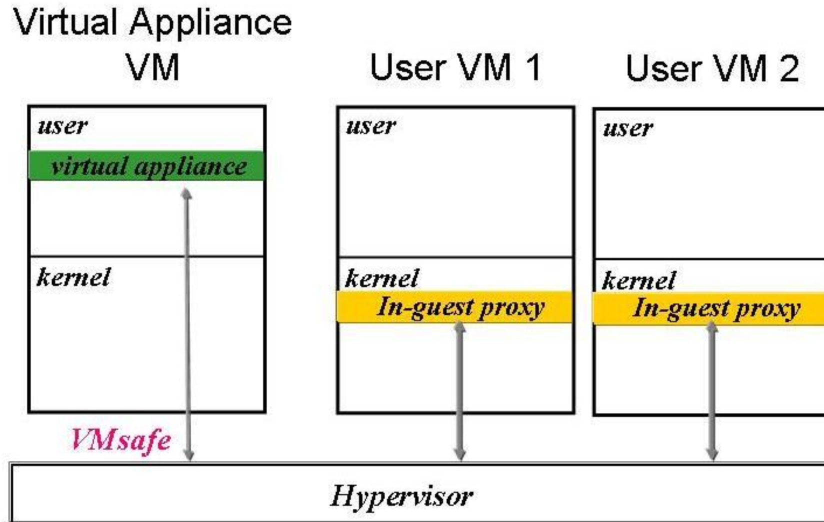

Figure 1: *In the virtual appliance execution architecture, common kernel services in user VMs are refectored to a user-level application running on the virtual appliance VM. However, to perform VM-specific operations, an in-guest agent needs to be deployed in each user VM.*

Many applications running on the virtual appliance may need to interact with the user VMs when certain events occur. For example, when an AV scanning virtual appliance application detects an AV string signature in a user VM's physical memory block, it needs to first identify to which process the memory block belongs, and then terminate the process if necessary. To find out which process owns which physical memory blocks and then terminate these processes require invoking systems service functions specific to the user VM in question, and thus cannot be directly initiated by the virtual appliance VM, which is outside the user VM. One way to address this problem is to install in each of the user VMs an in-guest agent that acts as a proxy which retrieves information from and exercises control over the user VM it is in. However, this approach entails two problems. First, installing an agent in a user VM takes additional efforts and has been a major pain point for data center administrators. This is why an agentless architecture is always preferred over an agentful one. Second, putting the agent of a security-related virtual appliance application into a user VM runs the risk that a kernel rootkit compromising the user VM may disable or tamper with the agent and thus indirectly cripple the associated virtual appliance application. This paper describes a stealthy agent deployment and execution mechanism called SADE that injects an agent into a user VM without any manual installation and invokes the agent's functions in a way that is nearly invisible to the user VM's kernel.

Conceptually, SADE is designed to inject a piece of arbitrary binary code into a VM's kernel, and then execute it within the VM's kernel address space in a way that does not require installing any new code in that VM. Moreover, other than calling kernel service functions, the execution of the injected code is invisible to and thus difficult to intercept by the VM into which it is injected. There are two main ideas in SADE. First, SADE supports an *out-of-VM* kernel code injection mechanism that replicates the main functionality of a kernel loader outside the VM whose kernel is the target of code injection, and therefore does not need to involve the VM at all, including its kernel loader. Second, SADE simulates an exception in the injected VM and hijacks the corresponding exception handler to invoke the execution of the injected code in the kernel context of the injected VM. To ensure stealthy execution, SADE enforces the following invariant: The injected code is visible only when it is invoked, which significantly decreases the window of vulnerability to tampering of SADE's injected code.

The rest of this paper is organized as follows. Section 2 reviews previous work on protecting kernel

code or data structures from being tampered by kernel rootkits. Section 3 describes the detailed design of SADE. Section 4 details a kernel memory scanning virtual appliance application that incorporates SADE as its in-guest agent deployment and execution mechanism. Section 5 presents a performance analysis of the current SADE prototype in terms of its start-up and run-time overhead. Section 6 concludes this paper with a summary of its research contributions.

2 Related Work

The basic techniques described in the paper to prevent tampering of in-guest components seem to be pretty straightforward: (1) using VM hardware to detect write attempts to protected pages, and (2) performing write target address comparison to enable byte-granularity protection (which is useful when a kernel driver is loaded into a region that is not aligned on a page boundary). The same techniques have been used in implementing software-based distributed shared memory systems in 1990s to enforce cache coherency, except in that case it is the kernel that monitors an application's address space, as opposed to the hypervisor monitoring a guest OS in this case.

Architectures that extract security software from within guest operating systems and place equivalent functionality within external hypervisor agents, custom security hypervisors, hardware components, or virtual machines dedicated to providing security for other virtual machines (so-called *security appliances* or *security VMs*) have recently received considerable attention. A key problem this work has identified, which is addressed in this paper, is the so-called *semantic gap* between high-level operating system abstractions and the raw memory-level access most of these solutions provide [1, 3].

A representative security hardware co-processor is Copilot [7]. An early system that extended an x86 hypervisor to investigate hypervisor security was Terra [2]. The Lares system provides an architecture for security code to safely executes inside a dedicated security VM, while the security VM itself can monitor other guest VMs by hooking selected locations with the monitored guests, in a completely transparent fashion supported by hypervisor extensions [5, 6]. Lares requies modifications to the underlying Xen hypervisor.

Another related area of work involves custom hypervisors that continuously verify that only trusted code is executed within guests. Examples of this work include Patagonix and NICKLE [4, 8]. Patagonix extends Xen while NICKLE enhances Qemu/Kqemu and VirtualBox. SecVisor is an example custom minimal-sized security hypervisor that continuously assures that only approved code is executed [9].This line of work differs from that described here in that our current work assumes execution of arbitrary, untrusted guest code. Our work takes advantage of the functionality of VMware's vmsafe-Mem/CPU API and does not involve any custom modifications to hypervisors, hardware, or guest operating systems.

3 Stealthy Agent Injection and Execution

3.1 Overview

A naive way to protect an in-guest kernel agent in the virtual appliance execution architecture from tampering is to write-protect the pages in which the reside. However, simply preventing a kernel agent from being written is not a satisfactory solution because it entails the following issues. First, if a kernel agent contains data, i.e., not just code, write-protecting the entire kernel module is not practical because one then needs to check if each write to the module's data area is legitimate or not. Second, if a kernel agent contains a function that the adversary can call to modify the agent's internal data state, then the problem is transformed into one that requires determining whether any calls to such functions are legitimate or not. Third, if a kernel agent's functions are called through a function pointer table residing outside the agent, the adversary can disable the entire kernel agent by modifying this function pointer table. This means one needs to write-protect

such function pointer tables as well. However, if a function pointer table can be legitimately modified at run time, one needs to determine if any writes to such function pointer tables are legitimate or not. Finally, if a protected kernel agent calls an unprotected kernel module, it may be possible for the adversary to modify the unprotected kernel module to perform malicious actions. For example, when a protected kernel agent calls an unprotected Ethernet driver to send confidential messages, the adversary could tamper with these messages by tampering with the Ethernet driver. However, the last issue is an instance of the general problem of "callees lying to callers" and is thus not inherent in the virtual appliance execution architecture. In this work, we will focus mainly on the first three issues.

Instead of taking a write-protection approach to ensure the integrity of in-guest agents, we take a stealthy injection approach to hide an in-guest agent from the injected kernel and thus minimize the window of vulnerability of the agent to malicious kernel rootkits. More concretely, we come up with the following set of requirements for an effective agent integrity protection mechanism:

- A protected kernel agent must be invisible to the rest of the kernel as much as possible. This way, a kernel rootkit is less likely to figure out the agent's data area or entry points to its functions, and to tamper with or directly call them.

- A protected kernel agent must not be invoked through some pointers residing outside the agent. Because no external pointers are needed to invoke a kernel agent, it is impossible to modify any such pointers to disable the agent.

- A protected kernel agent must be minimal in size to limit the attack surface and generic in functionality to permit sharing among different virtual appliance applications and reduce the total number of agents required in a VM.

For the virtual appliance execution architecture, the ideal in-guest kernel agent for a user VM is a minimal proxy that exposes every kernel service function in that VM as a RPC (remote procedure call) function that an application running on a virtual appliance VM can call directly. By "kernel service function," we mean all the service functions that the core kernel exports to kernel drivers for such functionalities as allocation of kernel memory, opening a device, querying a process's status, etc. With such an in-guest kernel agent, most of a virtual appliance's logic resides within the virtual appliance application. Moreover, a single in-guest agent in a user VM can service multiple distinct applications running on the virtual appliance VM.

However, the idea of using a minimal kernel service function proxy as a universal in-guest agent has one drawback: When a kernel service function takes a call-back function as an argument, it is difficult if not impossible for a virtual appliance application to register a call-back function, which supposedly should run in the virtual appliance VM, with the kernel service function proxy and reliably trigger the execution of the call-back function from the injected VM's kernel. Furthermore, the minimal kernel service function proxy architecture also complicates the accesses to input arguments that are passed as pointers to data stored in the virtual appliance VM.

An alternative to a minimal kernel service function proxy, each application running on the virtual appliance VM could have its own in-guest agent. Because the Windows kernel is close-sourced, there are significant implementation challenges to hide these application-specific in-guest agents from the guest VM into which they are injected, if the guest VM runs the Windows kernel. In the following subsections, we describe how to stealthily inject and execute an arbitrary piece of binary code into a Windows XP-based user VM from a Linux-based virtual appliance VM.

3.2 VMSafe Application Programming Interface

The current SADE prototype is built on top of VMware's ESX server, and leverages a new virtual machine control API called *VMSafe* (VMware Security API Framework), which is specifically designed to facilitate

the development of virtual appliances that need to access or control the state of other VMs running on the same physical server. VMsafe consists of a set of components and SDKs that support third-party security add-ons for VMware's vSphere environment. Instead of a single technology, the original version of VMsafe consists of the following 5 components, each of which corresponds to a particular SDK:

- *VMsafe-Net* enables a VM to perform TCP/IP packet-filtering for another VM, and provides the capability of specifying a fast path filter to be executed by the hypervisor and of moving packet filtering state associated with a VM as it is migrated between physical machines.
- *VMsafe-Mem/CPU* enables a VM to take control when a certain page-level virtual memory event occurs at another VM, e.g., when a page is written or executed, and to externally inspect and modify the memory and CPU state of another guest VM.
- *Vsafe internal handlers* enables a VM to set an unlimited number of invisible breakpoints on another guest VM so that control is transparently transferred from a specified guest VM kernel addresses to a handler in the same guest and back.
- *Vsafe-VIPER* enables a VM to inject a piece of code into the address space of a process running in another VM to continuously and transparently monitor and modify the injected process code as it executes.
- Vsafe virtual disk enables a VM to navigate VMware's VMFS file system and scan all formats of VMware's VMDK files, which means that files of a VM can be scanned even when the VM is offline.

However, more recent versions of VMsafe focus only on the first two components. The VMsafe API opens up the control of one VM to another VM and makes it possible to develop various add-on applications that would not have been possible previously without access to the hypervisor. Similar APIs that expose such control of one VM to another VM are being developed by other HAL hypervisor vendors, such as Citrix, Microsoft, and IBM.

3.3 Stealthy Agent Code Injection

To inject a piece of binary code C into the kernel address space of a user VM requires solving two problems: (1) allocation of a kernel memory region to hold C, and (2) resolving all the references in C to kernel service functions. SADE effectively addresses these two problems and runs on the virtual appliance VM.

Instead of relying on the kernel loader, SADE incorporates a mechanism that automatically discovers the export table of a running Windows XP kernel image and then reconstructs the entry points of all the service functions that the kernel exports. More concretely, before a user VM starts up, SADE sets an execution trigger on the user VM. This execution trigger is fired when the EIP of this user VM first falls in the upper 2GB of the address space during its boot-up. The address at which the execution trigger is fired corresponds to the Windows kernel's base address. Once the trigger is fired, control is transferred to SADE, which then scans from the trigger point towards the lower end of the kernel address space to look for a specific PE signature at the end of every page (the start of a PE image is always page aligned). Once the kernel image's PE signature is located, SADE homes in to the export table, and parses the table to construct a map for the entry points of all exported kernel service functions. Because SADE can only scan a user VM's guest physical pages, it needs to use the VMsafe API to translate the virtual addresses of the guest virtual pages being scanned into their corresponding guest physical addresses.

To place an in-guest agent binary into a Windows-based user VM, SADE needs to invoke a kernel service function `ExAllocatePool` to allocate a piece of memory from the user VM's non-paged pool to

hold the agent binary. However, the code to call the `ExAllocatePool` call itself needs to be injected first before it can be executed. Therefore, SADE needs a bootstrap mechanism to invoke the memory allocation code that calls `ExAllocatePool`.

Let's call the function whose goal is to allocate a piece of kernel memory A, into which a virtual appliance V places a suspended user VM G to allocate a kernel memory region from G. The detailed steps that SADE uses to load A into G and execute it are:

1. Copy a guest physical memory page P of G to V, and copy A into P.

2. Copy the first few memory locations pointed to by G's "invalid op code" interrupt descriptor table (IDT) entry to V and replace it with a jump instruction to the beginning of P.

3. Copy the memory location pointed to by G's EIP to V and replace it with an instruction with an invalid op code.

4. Schedule G to run next.G immediately encounters an "invalid op code" exception because its EIP points to an invalid instruction, and control jumps to A.

5. A calls `ExAllocatePool` to allocate a kernel memory region in G, returns the region's address space range to V, and transfers control back to V.

6. V copies the actual in-guest agent to the allocated address space range in G, and restores P, the memory locations pointed to by G's invalid op code IDT entry and G's EIP to their original contents.

In Step (1), we assume the page P is never used during the call to `ExAllocatePool`. The page containing the kernel image's PE header meets this requirement. However, there is no guarantee that this is always safe. Instead, the current SADE prototype uses the following technique to obtain a piece of kernel memory that is guaranteed to be unused. SADE parses the function body of `ExAllocatePool` to locate its return instruction, and sets an execution trigger at the page containing this return instruction. As the user VM boots up, the first time when its kernel calls `ExAllocatePool` and is about to return, the execution trigger is fired and SADE takes control. At this point, a kernel memory area has been allocated but not yet used. So SADE "hijacks" this kernel memory area by copying its memory-allocating bootstrap code (i.e., A) to it. After the bootstrap code completes its execution, control goes back to the original caller of the hijacked `ExAllocatePool` function. This technique assumes that the `ExAllocatePool` function is called at least once during a VM boot-up. In practice, the `ExAllocatePool` function is indeed called multiple times during the system start-up.

Although the VMsafe API allows V to directly modify the EIP of G, it is not considered safe to do so, because at the time when V is to invoke an in-guest agent in G, the current EIP of G may point to an instruction of a user process and its associated execution context (e.g. stack and heap) may not be appropriate for the execution of the injected in-guest agent or the bootstrap code. Instead, in Steps (2), (3) and (4), SADE modifies the entry point of the handler for the invalid op code exception, and intentionally triggers an invalid op code exception to transfer control to the injected code, which is supposed to run inside the kernel.

In Step (4), to avoid the double fault problem, when control reaches A, it first returns from the invalid-opcode exception handler to another piece of code in P, and that piece of code in turn calls `ExAllocatePool`.

In Step (5), to transfer control from A to V, SADE uses VMsafe's breakpoint feature by allocating a special page that is guaranteed not to be used, setting a write trigger on that page, and at run time intentionally writing to that location to transfer control back to V.

3.4 Stealthy Agent Code Invocation

Each in-guest agent is compiled as if it is a standard Windows kernel driver. Given an agent binary, SADE performs the necessary linking using the kernel service function map derived in the boot-up phase, and relocates the base addresses in the agent binary accordingly once the memory region used to the agent binary is determined.

Once the binary code for an in-guest agent X is ready and the kernel memory region to hold it in a target user VM is known, the virtual appliance copies the agent code into the allocated region. Later on, when a certain event in this user VM occurs, the virtual appliance invokes the in-guest agent by using the same control transfer mechanism shown in Steps (2), (3), (4), (5) and (6). Similarly, control transfer from the in-guest agent back to the virtual appliance is also through a write trigger.

Suppose at the time when X is injected into G, the EIP of G points to an instruction of a user process U. There are two possible outcomes for X's execution. First, suppose X calls a kernel service function that eventually terminates the process U. In this case, the kernel of G will schedule another ready process to run once the hypervisor schedules G to run. Second, suppose X completes its execution and returns, control should be returned to SADE and then to the interrupted EIP of U. However, because SADE already returns from the invalid-opcode handler *before* calling X to avoid the double fault problem, it has to use the `IRET` instruction with proper kernel stack set-up to transfer control from the kernel to U.

When SADE injects an in-guest agent binary, it includes a wrapper function around the agent binary to perform the following functions:

- Before calling the in-guest agent, changing the FS register's content from $0x3B$ to $0x30$ if the interrupted EIP points to an instruction of a user process.
- After calling the in-guest agent, writing to a special memory page to invoke a write trigger and transfer control to SADE.
- After SADE transferring control back to G, using `IRET` to return control to the interrupted EIP if it points to an instruction of a user process.

When a SADE-induced invalid opcode interrupt occurs, the associated interrupt handler *returns* to the wrapper function, which in turn calls X. As a result, when X is done, control naturally is returned back to the wrapper function.

One of the kernel memory pages allocated during the boot-up is dedicated to input and output arguments. The virtual appliance puts the input arguments into that page, and the in-guest agent returns outputs via the same page.

To protect the injected in-guest agent from unauthorized modification, SADE sets a write trigger on the agent's code area. To further protect the injected code from being read, SADE could optionally zero out an in-guest agent's in-memory image when a call to the agent's functions is completed, and restore the agent's image when control is about to be transferred back to the agent. This way, a VM's in-guest agent's image is visible to the VM's kernel only when its functions are being executed.

Through the combination of the above techniques, SADE enables a VM to inject and start the execution of a piece of code in another VM without involving the target VM's kernel. In addition, the execution of the injected code does not leave any side effects except those left by its calls to kernel service functions.

Although SADE minimizes the window of vulnerability of injected in-guest agents, it does not completely remove it. A kernel rootkit can potentially examine an injected in-guest agent by intercepting the kernel service functions the agent calls, or by hijacking some exception handlers and waiting for the called kernel service functions to trigger the corresponding exceptions.

4 Application: VMSlayer

To test the effectiveness of SADE, we have built a security appliance prototype on the VMware ESX platform using the VMware API. This security VM monitors other guest VMs, which all run Windows XP. Although Windows XP is typically a client operating system, it is uncommon in modern data centers that multiple Windows XP clients are consolidated onto a single server.

The security appliance prototype consists of the following components:

- A security virtual appliance VM. This VM is created by installing the VMsafe SDK into an unmodified Centos 5.2 Linux system running in a suitably configured VM running top of VMware ESX hypervisor. This VM is configured without a GUI and with 256 Mbytes of memory.

- A user-level memory-scanning security application. This application is a normal Linux C program written using Pthreads (Posix Theads) and linking with the VMsafe SDK library. It includes a string signature matching module ported from an existing Symantec security product, and uses a string signature database that is loaded entirely into memory at start-up time.

- An kernel-level in-guest agent. This was a custom-made Windows XP kernel driver, which is either explicitly loaded into the kernel when the guest VM starts up or injected into the guest VM using SADE.

The security VM runs first, after the physical server is booted up. When the security VM itself boots Linux, the security application is run as a standard Linux server, and registers specific VMsafe events using a pre-defined VMsafe network address and port. Thereafter it will receive from the hypervisor notification of specified events in monitored VMs.

The configuration information describing a guest VM indicates that it is to be monitored by the security VM. When a monitored VM is powered on, the security VM is notified, and creates threads and states to track the monitored VM. In addition, it creates mappings that describe to the hypervisor how accesses to selected ranges of the guest physical address space of the monitored VM are to be handled by the hypervisor.

The goal of our security virtual appliance is to scan the guest physical memory of other VMs to check if they contain any string signatures. To do so, the security virtual appliance needs to detect guest physical pages that are modified and subsequently executed. More specifically, on the first attempt to execute code in a page of a monitored VM, if the exact content of the page has not been previously scanned by the security VM, the hypervisor halts the virtual CPU whose control is about to enter the page and the page is mapped through to appear in the address space of the security application in the security VM. The hypervisor then signals the event to the security application, resulting in execution of the security application's thread associated with the monitored VM. This security thread typically obtains access to not only the contents of the physical page that contains code about to be executed, but also to the physical page that appears next in the virtual address space of the monitored VM. Having access to both pages is necessary because byte sequences crossing page boundaries may match string signatures. The security application then invokes the string signature matching to scan these two pages. Normally, upon completion of such a scan the security application signals to the hypervisor, via the VMsafe API, that the page has passed inspection and can be executed, and execution of the interrupted virtual CPU in the monitored VM resumes.

Although this process somewhat resembles a very heavy page fault, in which the security VM is executed as part of page fault handling in the monitored VM, it is different in that the security VM is invoked only when control is first transferred to a page after it is modified. Thereafter, further execution of the unmodified page will not invoke the security VM. If the page is modified again, however, the next attempt to execute code in the page will again trigger the execution of the security VM.

If the page inspected by the security VM is found to contain a string signature suggestive of malware, it is essential to trigger various types of specific remediation and other actions in the monitored VM. However, it is not desirable to simply convict the monitored VM and kill it. Instead, the security application should pinpoint the culprit process owning the signature-containing page. To this end the security application relies on the in-guest agent to perform such VM-specific operations as identifying a process owning a particular physical page.

The security application and the in-guest agent share a private communication mechanism based on VMsafe's page mapping capability. This communication mechanism is bidirectional. It is synchronous from the point of view of the in-guest agent, but asynchronous from the point of view of the security VM. This communication mechanism is used constantly from the moment when the in-guest agent starts execution. Run-time information regarding all processes executing on a particular monitored VM is exchanged with the security VM through this communication mechanism. Using this communication mechanism, the security VM issues commands related to particular processes on the monitored VM, such as kill a process or obtain additional information about a process. The in-guest agent can also issue commands to functionality that has been offloaded to the external security VM.

In our prototype the security VM somewhat verifies the within-guest driver's execution rate and execution state by constantly executing a challenge-response communication cycle with the within-guest agent. Potentially the security VM can protect, verify, and re-load, if necessary, the code to within-guest agents.

Using VMsafe, implementation of the memory-scanning virtual appliance prototype was relatively straightforward. The in-guest agent is able to obtain low-level operating system state at a fairly fine granularity, e.g. the contents of the process list. We were able to exchange this sort of low-level OS state with the security VM in near real time. In the end, the security VM is able to scan pages of monitored guest VMs to identify attempts to execute real-world malware code, and invoke the corresponding in-guest agent for remediation.

The experience of implementing this memory-scanning virtual appliance strongly suggests that the idea of refactoring common kernel services, in this case, physical memory page scanning, in a virtualized server into a separate virtual appliance is indeed feasible, and use of in-guest agents can successfully solve the semantic gap problem. When code and data are mixed in the same page, tracking page status using VMsafe may require additional software inspection. Thus performance can be improved if packing code and data into the same page is avoided. The user-level application running on the virtual appliance must be carefully written as multi-threaded soft real-time code. In our case, the entire design, including the memory scanning engine, must be such that no lengthy delays ensue. In addition, the security appliance must handle concurrent scans on behalf of all virtual CPUs of all monitored VMs. Finally, one likely software engineering advantage of the virtual appliance architecture is that compared with kernel drivers residing in monitored VMs, implementing the same functionality in a user-level application on a virtual appliance VM is more straightforward and flexible. For example, in our memory-scanning virtual appliance, it is easier for the security application to maintain a large string signature database in memory than its kernel driver counterpart.

5 Performance Evaluation

There are two versions of VMSlayer, each corresponding to a different implementation of the in-guest agent deployment and execution mechanism. This section presents the performance overhead of these two implementations. The hardware used for these experiments consisted of an HP Proliant DL360 G5 server with 2 3GHz Intel Xeon E5450 quad-core CPUs (for 8 CPUs total) and with 16 GB of memory. SAN storage was provided by a DELL EquaLogic PS3000 iSCSI server with 4 TB of disk storage. The HP server ran VMware ESX 4.0.0. build-133495, the user VMs being monitored ran Windows XP Professional SP2 as

Step	Time (msec)
Recognize kernel export table	17
Install stub for `ExAllocatePool`	1
Execute stub for `ExAllocatePool`	1.1
Relocate and load in-guest agent	2

Table 1: *The additional overhead of SADE to deploy an in-guest kernel agent in a monitored VM during its start-up process*

their guest operating systems, and the VMslayer VM used a Centos 5.2 2.6.18-92.el5 Linux guest operating system.

The first version of VMSlayer uses SADE, which inserts an in-guest agent into a monitored VM at the time when the security VM is about to invoke a function In the in-guest agent. When a monitored VM starts up, SADE first takes control when the EIP reaches the kernel's base address, and it scans the kernel address space to locate the kernel export table. This step takes approximately 17 msec, as shown in Table 1. Then it installs an execution trigger at the return instruction of the `ExAllocatePool` function in the monitored VM. When the monitored VM's first call to `ExAllocatePool` is about to return, SADE takes control the second time, it takes about 1 msec to put into the monitored VM a bootstrap stub and remove the execution trigger associated with `ExAllocatePool`, and another 1.1 msec to execute the stub, which calls `ExAllocatePool` to allocate a kernel memory area for the in-guest agent. When the execution of the bootstrap stub ends, it returns to the security VM the base address of the allocated kernel memory area, together with the kernel export table. At this point, the security VM takes control the third time, links the in-guest agent's import table with the kernel export table, and relocates the in-guest agent's binary using the returned base address. This step takes 2 msec. So the total additional overhead to set up an in-guest agent during a monitored VM's start-up process to is about 21.1 msec. Note that the in-guest kernel agent has not been injected into the monitored user VM yet.

Step	Time (msec)
Invalid opcode exception triggering and handling	4.7
Ring3 to Ring0 Context Switch	0.1
In-guest function execution	1
Ring0 to Ring3 Context Switch	0.1
Return to Interrupted Program	1.9

Table 2: *The additional overhead of SADE to install an in-guest kernel agent, invoke one of its functions, and return control to the interrupted program.*

When the security VM in VMSlayer detects a signature match in a code page of a monitored VM, it starts the remediation action by installing an in-guest kernel agent and invoking a remediation function in it. The first step is to copy the in-guest agent into the pre-allocated kernel memory area, modify the IDT entry associated with the invalid opcode exception handler, trigger an invalid opcode exception, and run the exception handler embedded in the in-guest agent, which in turn restores back the IDT entry and calls a wrapper function in the in-guest agent. This first step takes 4.7 msec, as shown in Table 2. The wrapper function modifies the FS register and calls the actual remediation function in the in-guest agent via an ring0 to ring0 `IRET` instruction. This takes 0.1 msec. Then it takes 1 msec for the remediation function to run,

which in this case identifies the process owning the matched code page, terminates the process, and returns to the wrapper function. Finally, the wrapper function takes 0.1 msec to restore the FS register, and returns control to the interrupted user program via an ring0 to ring3 `IRET` instruction, which costs 1.9 msec. So the total time required to invoke and complete each mediation action in VMSlayer is 7.8 msec.

In the second version of VMSlayer, an in-guest agent is a standard Windows kernel driver that has to be loaded into the kernel as a boot driver before any user-level processes start. Note that loading a kernel driver into a VM requires installation of a user program on that VM first, and therefore is not "agentless." The in-guest agent includes two kernel threads: a writer thread that informs the security VM, on which SADE runs, of process start/stop events, and a reader thread that polls a shared memory page for remediation commands from the security VM. When the security VM detects a code page in a monitored user VM matching a virus signature, it puts the active thread of the infected process into an infinite loop, and invokes an in-guest agent function by setting a special shared memory variable and inserts input arguments into the monitored user VM, e.g. the ID of the infected process. Once the in-guest agent's polling thread is scheduled and recognizes that the memory variable be set, the in-guest agent invokes the requested function with the inserted input arguments.

On average it takes 8 msec for this version of VMSlayer to invoke the same remediation function, but the measured delay of each invocation varies vary widely, from 2 sec to 26 msec, because of the non-deterministic delay associated with thread scheduling.

In summary, even though SADE needs to dynamically install an in-guest agent every time when it invokes a function in it, the average end-to-end per-invocation delays of these two versions of in-guest agent deployment and invocation mechanisms are actually comparable. This shows that the more flexibility and stronger security of SADE does not entail any additional performance overhead.

There are two reasons why the agentless SADE does surprising well when compared with the vanilla agentful architecture. First, the agentful architecture is asynchronous in that the in-guest agent needs to poll for a remediation command from the security VM. In contrast, SADE is synchronous and incurs almost no additional latency. Second, the polling thread of the agentful architecture operates in a different process context than the process triggering the security event,, so it needs to obtain a process handle to the triggering process before being able to manipulate the process. Obtaining a process handle incurs additional overhead because it requires access control checks by the security monitor. In contrast, the dynamically installed agent in SADE almost always operates in the same process context as the triggering process, and therefore avoids the additional step to obtain its process handle.

6 Conclusion

The proposed virtual appliance execution architecture is a promising way to remove redundant resource usage in future data centers, where a single physical server can host dozens or even hundreds of virtual machines. A key design decision in this virtual appliance architecture is how to enable a virtual appliance to perform VM-specific operations, such as accessing and modifying kernel service functions exported by a VM's kernel. An agentless architecture is more secure, less flexible and easier to deploy, whereas an agentful architecture is more flexible, less secure and more difficult to deploy. This paper describes the SADE system that simultaneously achieves the flexibility of the agentful architecture and the security and ease of deployment of the agentless architecture. The key building blocks of SADE are its ability to resolve calls to kernel service functions without relying on the kernel loader, and its code injection technique that ensures the injected code is invoked in a proper execution context. We have built a fully operational SADE prototype using the VMsafe API and applied it to a security-based virtual appliance application, which scans the kernel memory of the VMs against a string signature database. Initial performance results suggest the latency penalty of SADE's code injection and deployment mechanism is relatively insignificant even with

full-scale tamper protection. In summary, the research contributions of this work are

- A stealthy code injection mechanism that allows one VM to inject binary code into another VM without involving the target VM or requiring pre-installation of any other code on the target VM.
- A remote code execution mechanism that allows one VM to run a piece of code on another VM using a proper execution context without requiring any help from the target VM.
- A fully working SADE prototype for a Linux-based source VM and a Windows XP-based target VM, and a demonstration of its usefulness through a complete virtual appliance application, a shared kernel memory scanner running on a virtual appliance VM that scans the kernel memory for other user VMs running on the same physical machine.

References

[1] Peter M. Chen and Brian D. Noble. When virtual is better than real. In *HOTOS '01: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, page 133, Washington, DC, USA, 2001. IEEE Computer Society.

[2] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: a virtual machine-based platform for trusted computing. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 193–206, New York, NY, USA, 2003. ACM.

[3] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 128–138, New York, NY, USA, 2007. ACM.

[4] Lionel Litty, H. Andrés Lagar-Cavilla, and David Lie. Hypervisor support for identifying covertly executing binaries. In *SS'08: Proceedings of the 17th conference on Security symposium*, pages 243–258, Berkeley, CA, USA, 2008. USENIX Association.

[5] B. D. Payne, M. D. P. de Carbone, and Wenke Lee. Secure and flexible monitoring of virtual machines. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 385–397, 2007.

[6] Bryan D. Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. Lares: An architecture for secure active monitoring using virtualization. In *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 233–247, Washington, DC, USA, 2008. IEEE Computer Society.

[7] Nick L. Petroni, Jr., Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, pages 13–13, Berkeley, CA, USA, 2004. USENIX Association.

[8] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *RAID '08: Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, pages 1–20, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 335–350, New York, NY, USA, 2007. ACM.