Semantics-Driven Introspection in a Virtual Environment

Francesco Tamberi, Dario Maggiari, Daniele Sgandurra
Dipartimento di Informatica
Università di Pisa
{tamberi, maggiari, daniele}@di.unipi.it

Fabrizio Baiardi
Polo G. Marconi, La Spezia
Università di Pisa
baiardi@di.unipi.it

Abstract

Semantics-driven monitoring discovers attacks against a process by evaluating invariants on the process state. We propose an approach that increases the robustness and the transparency of the run-time monitoring system by introducing two virtual machines (VMs) running on the same platform. One VM runs the monitored process, i.e. the process \( P \) to be protected, while the other one evaluates invariants on \( P \) state each time \( P \) invokes a system call. To this purpose, an Introspection Library allows the monitoring VM to access the memory and the processor registers of the monitored VM.

After describing the overall architecture, we focus on the Introspection Library and the problems posed by the introspection of variables in the memory of a program running in a distinct VM to evaluate invariants. A first prototype implementation is also presented together with preliminary performance results.

1. Introduction

Virtualization is becoming increasingly popular, because enterprises can reduce the total number of servers by migrating the real environments, hosted on physical machines, to virtual machines (VMs) running on a single server. Since a VM is an exact replica of a real machine, it can execute the same OS and applications but several VMs can share a single physical host concurrently. The consolidation of multiple server onto a single platform harnesses unused computing power and cuts hardware and lifetime costs.

While these advantages cannot be neglect, we believe that virtualization should also be exploited to build more robust systems [3]. As an example, the Virtual Machine Monitor (VMM), i.e. the software that runs, confines, and manages the VMs, exports a control interface that allows a monitoring VM to analyse the running state of other VMs through Virtual Machine Introspection (VMI) [4]. The introspection capability leverages the fact that a VM completely encapsulates the state of the corresponding physical host, so that a monitoring VM can analyse in full detail the host running inside another VM, for example it can search for specific values in the memory of another VM or inspect the content of the VCPU registers. In this way, a monitoring VM can analyse and compare the data obtained through introspection against those returned by invocations to the OS of a monitored VM. Any discrepancy signals that an attacker may have altered the OS of a monitored VM [1].

This paper proposes an approach to discover attacks against a process \( P \) based upon the evaluation of an invariant on the state of \( P \) every time \( P \) invokes a system call. The approach requires a static tool, which analyses \( P \) program to deduce an invariant to be evaluated for each system call, and a run-time tool that intercepts any system call issued by \( P \) and evaluates the corresponding invariant. In this paper we focus on the run-time tool, which is implemented by two VMs: a Monitored VM (Mon-VM), which runs \( P \), and a monitoring Introspection VM (I-VM). The I-VM runs an Assertion Checker that evaluates invariants on the state of \( P \). The Assertion Checker accesses the values of variables of \( P \) and the processor registers of the Mon-VM through an Introspection Library. Every time \( P \) issues a system call, the Mon-VM transfers control to the I-VM, which: (i) retrieves the value of the processor registers of the Mon-VM; (ii) determines the invariant \( I \) paired with the system call that \( P \) has issued; (iii) retrieves the values of variables of \( P \) that \( I \) refers to; (iv) evaluates \( I \) and kills \( P \) anytime \( I \) is false. In this way, the monitoring is both fully transparent to \( P \), which is not aware of being monitored, and highly robust, because it is applied by a VM that is separated from the one that executes \( P \).

The rest of the paper is organised as follows. Section 2 briefly outlines some features of Xen memory management, which we have exploited to implement the current prototype. Section 3 describes the overall architecture of the run-time tool, the Introspection Library and the Assertion Checker, and evaluates the overhead of the current prototype. Section 4 discusses related works. Finally, Sect. 5 draws some conclusions and outlines future developments.
2. Xen Memory Management

This section recalls some features of Xen memory management that influences the overall architecture of the run-time tool. Xen [2] is an open source VMM, or hypervisor, which supports the virtualization of machine hardware resources and their dynamic sharing among OSes running inside several VMs. Xen adopts a para-virtualized approach where OSes need to be modified to run on top of the VMM. However, by exploiting the hardware support for virtualization [6], Xen can also execute unmodified guest OSes.

To deal with memory virtualization, one of the most complex tasks for a hardware-level VMM, Xen considers three distinct issues: (i) physical memory management, e.g. how to avoid memory fragmentation; (ii) virtual memory management, e.g. how to minimize the overhead introduced when a VM is scheduled; (iii) page table (PT) management, e.g. how to validate each memory access to satisfy the isolation requirement among VMs. To give the guest OSes the illusion of a contiguous address space, Xen defines two distinct address spaces: Machine memory, i.e. the total amount of physical memory of the host that runs Xen, and Pseudo-Physical memory, i.e. the set of addresses as seen inside a VM. The mapping between the two spaces is implemented through two tables: Machine-to-Physical (M2P), which maps the real memory pages into pseudo-physical pages, and Physical-to-Machine (P2M), one for each domain, which implements the reverse mapping. To minimize the performance degradation of VM context switching due to TLB misses, the top most 64MB (for 32 bit architectures) of the virtual address space of each process contains a mapping for the Xen hypervisor itself. As far as concerns the management of PTs, there are two possible solutions: (i) shadow PTs, which require that a guest OS implements virtual PTs that are not visible to the MMU; (ii) guest OSes directly manage PTs that are read-only so that an OS has to invoke Xen through hypercalls to update the mapping. In the case of shadow PTs, to prevent each VM from interfering with another one, Xen traps each access to the virtual PTs and propagates their updates to the real PTs used by the MMU.

3. Architecture of the Run-Time Tool

Our semantics-driven approach detects attacks against a process \( P \) by evaluating invariants, i.e. assertions, that constrain the value of some \( P \) variables, at each system call invocation. System call sites are among the most appropriate choices for program points to check variable values, because these are the points where the monitored system switches from user-level to kernel-level. Therefore, this is the most critical time when an attacker may exploit some vulnerability to gain control of the system. Invariants are the output of a static analysis of the program source code. To be fully integrated with the run-time tool, the static tool should generate an invariant for each system call that relates values of programs variables and of system call parameters. Then, this invariant is paired with the PC value when \( P \) executes the system call.

The architecture of the run-time tool includes two distinct VMs to increase robustness with respect to attacks against the monitoring system. These two VMs are the Monitor VM (Mon-VM), which executes the monitored process \( P \), and the Introspection VM (I-VM), which verifies the integrity of \( P \). This VM runs an Assertion Checker process that evaluates invariants on \( P \) state and accesses its variables, in the memory of the Mon-VM, through VM introspection. The input of the Assertion Checker is a set of invariants of the form:

\[
\{ \text{PC}, \{ \text{var name: addr: type}, \{ \text{expr on vars} \} \}
\]

where: \( \text{PC} \) is the program counter paired with a system call, \( \{ \text{var name: addr: type} \} \) is a set of variable names, their virtual address and their type, \( \{ \text{expr on vars} \} \) is a set of relations among variables of the previous set.

The kernel of the Mon-VM transfers control to the I-VM every time \( P \) invokes a system call. The I-VM freezes the execution of the Mon-VM and the Assertion Checker exploits the Introspection Library to retrieve the current PC of \( P \) and the values of the variables of \( P \) that the invariant paired with the current PC refers to. If the invariant is false, the I-VM kills \( P \), otherwise it resumes control of the Mon-VM from the current PC.

3.1. Introspection Library

The Introspection Library is invoked by the Assertion Checker whenever \( P \) issues a system call. The library implements two introspection functions, namely Memory Introspection, to access the memory of a monitored VM both at the user and at the kernel level, and VCPU-Context Introspection, to retrieve the state of the Mon-VM virtual processor.

To implement user-space memory introspection, the Introspection Library needs to access any physical memory location allocated to the Mon-VM that corresponds to a virtual address of \( P \). To translate a virtual address, the Introspection Library directly accesses the PTs of \( P \) and then follows the pointer to walk the paging levels to retrieve the pairing between a virtual and a physical address. In the case of para-virtualization, the addresses in all the page levels and in the registers of a virtual context of a VM are machine addresses. This implies that the Introspection Library has to map three pages to translate a virtual address into a machine address and map the corresponding page.
Conversely, Xen manages static addresses in a para-virtualized OS, such as those paired with the kernel exported symbols, as pseudo-physical addresses. To this end, when Xen starts a VM, kernel static addresses are relocated, and the original addresses are managed as pseudo-physical ones. For this reason, to translate a pseudo-physical address \( PPA \) paired with a kernel symbol \( KS \), the Introspection Library executes the following steps: (i) translate \( PPA \) into a machine address \( MA \) using the P2M table; (ii) request Xen to map the page at the base address of \( MA \) and retrieve from the resulting offset the relocated pseudo-physical address \( PPA_2 \) of the kernel symbol \( KS \); (iii) access the P2M table to translate \( PPA_2 \) into the corresponding machine address \( MA_2 \); (iv) request Xen to map the page at the base address of \( MA_2 \) into the address space of the Assertion Checker process. This page stores the kernel data structure pointed to by the kernel symbol \( KS \). As soon as the Introspection Library has mapped the page that stores the pointer to the kernel page directory, referenced to by the swapBufPg_dir symbol, it can translate pseudo-physical addresses by accessing the kernel PTs in the same way of a process virtual address. In this case, the Introspection Library maps three pages instead of executing the previous four steps. Finally, when exploiting processor virtualization extensions, Xen applies the shadow PTs mechanism and both the page directory and PTs store pseudo-physical addresses. Each time a PT needs to be updated, Xen propagates the update to the real PT, which is known to the MMU. To this end, the Introspection Library exploits the Xen page_array structure, which stores the pairing between pseudo-physical frame numbers and machine frame numbers. To optimise the translation of variables allocated in the same page, the Introspection Library implements a TLB-based software solution to keep track of the pairing among virtual addresses and machine addresses for the monitored process.

The VCPU-Context introspection allows the Assertion Checker to monitor, and modify, the content of any Mon-VM register. To support the context switch between two VMs, Xen saves the values of the CPU registers in a Virtual CPU-Context paired with each VM. When a VM is going to be scheduled, the current values of the registers are saved into the VCPU context of the running VM, while the registers of the new VM are restored from the proper VCPU context. To evaluate the invariants, the Assertion Checker exploits the VCPU-Context introspection capability of the library to retrieve the current PC of \( P \) and to map the pages storing the variables of \( P \) into its address space to fetch their values.

At runtime, when \( P \) issues a system call, control is hijacked and transferred to the I-VM, where the Assertion Checker: (i) reads the current PC and the current value of the variables to evaluate the invariant paired with PC; (ii) evaluates the invariant. In more detail, the Assertion Checker (see Fig. 1): (i) accesses the VCPU context to read the kernel_sp register, which points to the top of the kernel stack; (ii) maps the kernel stack; (iii) reads the value of the ESP register, which points to the base of the user stack; (iv) maps the user stack; (v) locates in the user stack of \( P \) the return address of the system call: since the offset of the return address from the stack pointer depends upon the system call type, the Assertion Checker reads the EBX register to identify the system call; (vi) after reading the return address, the Assertion Checker maps the pages storing the variables paired with this return address; (vii) reads the value of the variables; (viii) evaluates the invariant. If the invariant is satisfied, the Assertion Checker resumes the execution of the Mon-VM, otherwise it kills \( P \).

### 3.2. Performance Results

The most time-consuming operation of an invariant evaluation is the time to access the variables. The average time to map a page of \( P \) into the Assertion Checker address space is about 50\( \mu \)secs. For each system call, at least two pages have to be mapped for, respectively, the kernel and the user stack. Moreover, at most one further page for each variable that the invariant refers to has to be mapped. Thus, the corresponding overhead is at least 150\( \mu \)secs, since each invariant refers at least one variable. By exploiting the software TLB, this time can be considerably reduced anytime several program variables are stored in the same page and the Assertion Checker can access the variables without mapping further pages of the virtual address space of \( P \). If the variables are stored in the same page, each access requires 20\( \mu \)secs. Therefore, since the software TLB can also be exploited to access the kernel stack, anytime the Assertion Checker needs to retrieve and evaluate the value of just one variable, the overhead due to the mapping and the evaluation is about 60\( \mu \)sec for each system call. Taking into account the rate of system call invocations, the overhead of the execution time is lower than 10% on the average.

### 4. Related Works

An approach to VM introspection is proposed in [8] with XenAccess, a monitoring library for OSes running on Xen, which provides virtual memory introspection and, as opposed to our library, virtual disk monitoring capabilities. Lares [9] is a security tool that can actively control an application running in a guest VM by inserting hooks into the execution flow of the process. These hooks transfer control to another VM that checks the monitored application using introspection and security policies. The guarded model [10] combines control flow and data flow analysis to generate and propagate invariants which guard system calls. [5]
presents an architecture model to protect host-based intrusion detectors by exploiting the confinement provided by a VMM to separate the IDS from the monitored OS. Xenprobes [7] is a framework to probe several Xen guest kernels simultaneously and that allows developers to implement their probe handlers in user-space.

5. Conclusion and Future Developments

We have proposed a semantics-driven approach to monitor program execution that exploits the virtualization technology to access the process memory and evaluate an invariant for each system call. We believe that this is a very general and selective strategy to detect attacks, because it can discover malicious updates to process code or data structures anytime a process requires a critical operation to the OS. Moreover, the isolation of the monitoring VM from the one that runs the monitored process increases the overall robustness.

An area of future research is the automatic extraction of invariants from the application source code of the monitored process. Since we are currently focused on the run-time aspects of the monitoring, to evaluate both its efficacy and efficiency, the deduction of invariants through a static analysis has been defined but not implemented yet. Moreover, we are currently working on a strategy that integrates static and dynamic information to compute the addresses of local variables, by keeping track of the value of the frame pointer.

References