A VMM-based System Call Interposition Framework for Program Monitoring

Bo Li, Jianxin Li, Tianyu Wo, Chunming Hu, Liang Zhong
School of Computer Science and Engineering,
Beihang University,
Beijing, China
{libo, lijx, woty, hucm, zhongl}@act.buaa.edu.cn

Abstract—System call interposition is a powerful method for regulating and monitoring program behavior. A wide variety of security tools have been developed which use this technique. However, traditional system call interposition techniques are vulnerable to kernel attacks and have some limitations on effectiveness and transparency. In this paper, we propose a novel approach named VSyscall, which leverages virtualization technology to enable system call interposition outside the operating system. A system call correlating method is proposed to identify the coherent system calls belonging to the same process from the system call sequence. We have developed a prototype of VSyscall and implemented it in two mainstream virtual machine monitors, Qemu and KVM, respectively. We also evaluate the effectiveness and performance overhead of our approach by comprehensive experiments. The results show that VSyscall achieves effectiveness with a small overhead, and our experiments with six real-world applications indicate its practicality.

Keywords—virtualization; system call interposition; VMM; program monitoring

I. INTRODUCTION
System call interposition is a powerful technique for regulating and monitoring program behaviors. It gives security systems the ability to monitor all of the application’s interaction with network, file system and other sensitive system resources. Many security systems, such as host intrusion detection systems (HIDS), leverage system call interposition to detect anomalous program behaviors. The discrimination between normal and abnormal behavior is based on what system calls are normally invoked by a running program. To guarantee the effectiveness and security of these security systems, system calls must be intercepted and handled safely and completely.

However, traditional system call interposition techniques [1][21][22] are usually implemented in operating system kernel, thus they are vulnerable to kernel attacks. For example, attackers can exploit kernel buffer flow to get full control over OS kernel, so as to bypass the interposition of system calls. And, most of them require modifying or recompiling the OS kernel, so they cannot support legacy applications and close-box operating systems.

To address the above problems, we intend to leverage virtualization technology to provide sufficient security and transparency. While, for virtual machine monitor (VMM), it can only “see” low-level events and states such as CPU registers and memory pages, which are not enough for security systems to understand the OS-level semantic. To bridge this semantic gap, we proposed a VMM-based system call interposition framework name VSyscall, which enable monitoring program behavior outside of the guest OS. The major contributions are summarized as follows:

- We design a VMM-based system call interposition approach which cannot be bypassed by guest OS even if guest kernel has been comprised.
- The method of system call correlating is proposed to establish the relations among related system calls. The relations are used by VSyscall to monitor and identify process behavior based on given patterns.
- A prototype of VSyscall system is designed and implemented for two different full virtualization environments, qemu [19] and kvm [20]. Malware samples and benchmark applications are used to evaluate the effectiveness and performance of VSyscall. Experimental results show its effectiveness with only incurring a small runtime overhead.

The remainder of this paper is organized as follows. Section II gives an overview of VSyscall architecture and describes the key techniques used in VSyscall. Section III describes two applications of VSyscall. Section IV evaluates VSyscall’s effectiveness and performance overhead. We discuss related work in Section V and conclude in Section VI.

II. VSYSCELL DESIGN AND IMPLEMENTATION

In this section, we describe the design of the VSyscall framework and illustrate our implementation approach to achieve our design goals.

A. Design Goals

The virtual platform consists of the underlying hardware, a VMM and a host OS. We assume each part of the virtual platform is trustworthy. A malware can compromise any part of the guest VM including the guest OS kernel, but it cannot subvert the underlying virtual platform. VSyscall has the following three design goals:

Effectiveness. VSyscall should be able to maintain its effectiveness and security even if guest OS kernel has been
The key to implement a binary open, read, ..., mmap, ... is the link between
arguments to registers, and then traps into the kernel using an interrupt instruction (e.g. INT 80h for Linux and INT 2e for Windows). Newer platforms, such as linux 2.6, Windows XP and later, normally use sysenter instruction to call system services.

**Binary Translation.** The key to implement a binary translation VMM is to capture the executing of sensitive instructions and handle them, since these instructions would interfere with the state of the underlying VMM or host OS [6]. The VMM will examine all instructions before execution. When a sensitive instruction is identified, it will be forced to trap into the VMM. INT n and iRET are sensitive instructions, so we can intercept them in the VMM. When a system call is invoked from the process in guest OS, it first traps into the VMM; here we insert a hook into the interrupt handler of the VMM to intercept system calls; when finished, the VMM will jump to the OS’s system call handler and let OS handle the system call; When the OS is finished, an IRET instruction will be executed which will also be captured by the VMM; Last, the VMM return control to the user process.

**Hardware-assisted Virtualization.** There are two mainstream hardware-assisted virtualization technologies: Intel VT and AMD SVM. System call interception is directly supported by AMD SVM; while for Intel VT, things become a bit more complicated, since INT n or sysenter/sysexit cannot be directly captured in the VMM. To solve this problem, we adopt the technique mentioned in Ether [7]. The key is to rewrite the system call entry address to an illegal value, so when a system call is attempted, a page fault occurs and a vmexit will be generated and delivered to the VMM’s interrupt handler, and we take integrity measurement here; when finishing the measurement, the origin address is restored to resume the normal execution.

**System Call Arguments Retrieval.** System call arguments contain key information which is very useful for security systems to identify malicious behaviors of a given program. We need to retrieve them in the VMM. Some arguments can be retrieved directly by accessing the VCPU context of the guest VM. For example, the system call number argument of Linux can be obtained by reading the EAX register of the VCPU data structure. Some arguments are stored on the user stack. We first get the user stack pointer through the relevant register, and then add the offset to get the actual address of arguments. It is important to note that for the reference arguments which we obtain in the VMM is a virtual address of the guest VM. To read the content of the virtual address, it needs a two-level address translation to get the actual machine address which can be accessed by the VMM. The first level of translation maps the guest virtual address into the guest physical address. The second level of translation maps the guest physical address into the machine physical address.

**D. System call correlating**

There are internal relations among System calls. For example, the loading of a DLL involves a sequence of system calls: {open, read, ..., mmap, ...}; open will return a file descriptor which is taken by mmap as an input argument. The return value of open is the link between open and mmap.

C. System Call Interception

In most modern operation systems, user processes which want to execute privileged operations must use system call interface to access kernel services. When a system call is invoked by a user level process, the process first places the
While in guest OS, programs are running concurrently, and another process may be running at the same time which also invokes system calls. So the system call sequence that VMM intercepts may look like this form: \{open(p1), open(p2), ..., read(p2), ..., read(p1), mmap(p1), open(p1)\}. open(p1) represents the open system call invoked by process1.

So given a system call, VSyscall must identify which process it belongs to. We use CR3 register to differentiate a process from the other. On x86 systems, CR3 register stores the base address of the table of Page Directory Entries (PDE) for the current running process, so it uniquely identifies a running process.

Next, we intercept the event of system call exiting to get the return value, since it is the link between the two related system calls. But the problem is, from the perspective of VMM, system calls are executed concurrently. And, the entering and exiting events of system calls are asynchronous. It means when we capture an entering event of system call in the VMM, the next exiting event we intercept may not be raised by the same system call. When invoking a system call, the instruction pointer (IP) of the running process will be saved by OS kernel; when the system call returns, OS kernel will load the saved instruction pointer to EIP register and resume the execution. Therefore, we use IP to determine whether an entering event and an exiting event belong to the same system call; if they have the same IP value, we conclude that they are within the same system call. Upon obtaining the return value of a system call, we can use this return value to establish the correlation between two system calls.

E. Process Behavior Monitoring

After intercepting the system calls, VSyscall analyzes them based on configurable patterns to monitor process behaviors. System call correlating is used to establish the relations between two system calls.

<table>
<thead>
<tr>
<th>Pattern: Kernel Module Loading</th>
</tr>
</thead>
<tbody>
<tr>
<td>open(path) \rightarrow fd</td>
</tr>
<tr>
<td>mmap(fd) \rightarrow addr</td>
</tr>
<tr>
<td>init module(addr)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VMM Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>open in: (CR3, IP) \rightarrow path</td>
</tr>
<tr>
<td>open out: (CR3, IP)=path and (CR3, fd) \rightarrow path</td>
</tr>
<tr>
<td>mmap in: (CR3, fd)=path and (CR3, IP)=path</td>
</tr>
<tr>
<td>mmap out: (CR3, IP)=path and (CR3, addr)=path</td>
</tr>
<tr>
<td>init module in: (CR3, addr)=path</td>
</tr>
</tbody>
</table>

There are two system call analysis modes: online and offline. For Offline mode, an offline analyzer is used to scan the system call sequence to match the specified pattern. For online mode, we take Linux kernel module loading as an example, and define a pattern which is used to get the kernel module’s name upon identifying its loading. The pattern is shown in Table I.

In this pattern, init_module system call identifies the loading of kernel module. The above pattern will be translated by VSyscall into a set of VMM actions, which is shown in Table II.

Fig. 2 shows the system call executing traces and the actions taken by VMM.

F. Implementation

VSyscall has been implemented for two virtualization platforms – Qemu and KVM. Currently, we target Linux as the guest OS and we implement VSyscall for x86 architecture to support all versions of Linux kernel. Note that VSyscall is not limited to Linux guest OS, and it can also be implemented to support other operating systems as long as they support system call mechanisms.

VSyscall for Qemu. We implement Vsyscall in Qemu-0.9.1. Linux system calls are usually invoked by INT 80h instruction, but new Linux 2.6 kernel also uses sysenter instruction to invoke some system calls. Our prototype disables SEP bit of CPUID in Qemu virtual CPU implementation, so every system call will be invoked using INT 50h. Then we insert a hook into QEMU to intercept INT80h instruction. The hook is placed into do Interrupt function. In the hook, we obtain system call arguments from virtual CPU’s registers which can be directly accessed through env data structure. We also use cpu_memory_rw_debug function provided by QEMU to read guest OS’s memory content given physical address as a parameter.

VSyscall for KVM. The kvm version of VSyscall is implemented in the kvm-84 and developed under the platform with Intel VT support. To intercept the system call, we first save the original value of register GUEST_SYSENTER_EIP and then overwrite it by an illegal address during the boot time of the guest OS. When a system call is invoked, it will incur a page fault and trap into the kvm kernel module. We first identify the system call invocation, then take integrity measurement, and upon finishing the measurement we restore the original
SYSENTER_EIP value to resume the normal execution. We obtain the system call arguments by reading the vcpu data structure. To access the virtual address in the guest OS, we first use gva_to_gpa function to convert the guest virtual address to guest physical address and then use kvm_read_guest to access the machine address to get the actual value.

III. APPLICATIONS

A. Integrity measurement

Integrity measurement [2][3][4][8][10] has been proposed and studied by many researchers as an effective way to counteract malwares and verify the integrity of computer systems. However, traditional integrity measurement systems required modifications to the OS kernel, some of them even need to modify applications [4], so they cannot support legacy applications and close-box operating systems. And, these systems are also susceptible to kernel attacks.

To solve the above problems, we implement a load-time integrity measurement system based on VSyscall. We interpret system call events invoked in guest OS from the hypervisor, identifies the system calls related to binary loading events, and infers executable’s path based on system call correlating method, then uses the path information to locate the corresponding binary’s image on disk and measure it. Some preliminary experiments have been conducted to evaluate this approach, and the results show its effectiveness.

B. Host-based Intrusion Detection System

System call interposition has been used by many Host-based Intrusion Detection Systems (HIDS) [21][22][23] to detect irregular and malicious behaviors. The accuracy of these systems relies heavily on the completeness and correctness of system call interposition. However, the interposition functions of these systems often suffer from kernel attacks. We leverage VSyscall’s interposition ability to intercept system calls which are then delivered to HIDS as its input. The pattern matching results of VSyscall will also be useful for HIDS for further analysis.

IV. EVALUATION

In this section we explore the effectiveness and performance overhead of VSyscall in each of our implementation environments.

A. Effectiveness

Kernel module loading is frequently used method for malware to enter and subvert Linux kernel. Adore-ng is a famous kernel rookit to infect Linux kernel as a loadable kernel module, so we choose it to evaluate the effectiveness of VSyscall in monitoring program behaviors. To mimic the behavior of a hacker, we replace nfs module with adore-ng, and then invoke modprobe command to load it into the kernel. The system calls intercepted by VSyscall are shown in Fig.3. CR3 is used to represent a process. The arguments contain key information which can be used by VSyscall to make decisions. For example, the mode argument of open system call represents the access mode of a file, and it can be used by VSyscall to determine whether to open it or not. As shown in Fig.3, three system calls, which are linked by return values, are highlighted and uniquely identify the loading of kernel module.

![Figure 3. The screenshot of VSyscall](image)

B. Performance

Our benchmarking tests were run on a Dell PC which is equipped with Intel Core2 Duo 2.4 GHz CPUs and 4G RAM. The PC runs the i386 version of the Debian Lenny Linux distribution as the host OS. We have evaluated VSyscall for guest OSes with Linux 2.4 and 2.6 kernels respectively, while we only show the results of Linux 2.6 due to limitations on space. Our experiments consist of microbenchmarks and application benchmarks, and all experiment results are average of ten iterations.

1) Microbenchmarks

We use lmbench tool [15] for our microbenchmark evaluation. Since VSyscall only affects the codes path of guest system call execution, we choose several microbenchmarks related with system call operations from lmbench to study the overhead of VSyscall. Table III and IV show our results for Qemu and KVM respectively.

First, we notice that system call handle time for vanilla KVM is much smaller than handle time for vanilla Qemu. This is because in Qemu the VMM must catch and emulate every guest system call instruction, while in KVM with hardware assistance a guest OS runs at privilege level 0, system call instructions can be executed without the intervention of VMM. Second, by comparing Table III and IV we can see that VSyscall-KVM incurs relatively more overhead than VSyscall-Qemu. The reason is that, in VSyscall for KVM, page faults are generated and trapped by VMM for intercepting guest system calls, which incurs significant overhead. While in Qemu, system call instruction

1 In fact, we can also obtain process name through VM introspection techniques
has already been identified and handled, so VSyscall can directly intercept guest system call in the interrupt handler of Qemu.

| Table III. MICROBENCHMARK RESULTS OF VSYSCELL-QEMU (µS) |
|----------------|---------|---------|---------|---------|
|                | null    | open    | read    | write   | Fork    |
| Vanilla        | 0.84    | 19.9    | 2.32    | 1.88    | 2881    |
| VSyscall       | 1.11    | 22.1    | 2.66    | 2.33    | 2949    |

| Table IV. MICROBENCHMARK RESULTS OF VSYSCELL-KVM (µS) |
|----------------|---------|---------|---------|---------|
|                | null    | open    | read    | write   | Fork    |
| Vanilla        | 0.17    | 1.66    | 0.26    | 0.24    | 2058    |
| VSyscall       | 5.74    | 14.21   | 6.19    | 6.11    | 2213    |

C. Application Benchmarks

To further evaluate the performance overhead introduced by VSyscall, we measure its runtime overhead on six application benchmarks.

Fig.4 shows the application benchmark results of VSyscall-Qemu and VSyscall-KVM, respectively. Overall, VSyscall-KVM incurs more overhead than VSyscall-Qemu, because it needs to trigger page faults to trap into the VMM for system call interception, which greatly affect the performance.

For file encryption and decryption applications, VSyscall-Qemu and VSyscall-KVM introduces 0.5% and 1.8% overhead on average, respectively. These two applications are computing-intensive, thus the overhead is small. File copy is I/O intensive task, which involves lots of read and write system calls, therefore its overhead is relatively higher than encryption and decryption. For kernel building application, the overhead of VSyscall-Qemu is negligible, while VSyscall-KVM is nearly 2%, which is more than 0.2% for VSyscall-Qemu. Note that the overhead here is a percentage value which represents the ratio of overhead time to normal boot time, and the boot time of guest linux in Qemu is much longer than the boot time in KVM. For virus scan application, both VSyscall-KVM and VSyscall-Qemu incurs negligible overhead.

V. RELATE WORK

Virtualization technology has long been used by security applications to enhance the security of computer systems [13][14][15][17]. The basic idea is to use VM to isolate these security applications from the protected systems, and leverage VMM to monitor and protect them without having to trust the operating system. Some research works focus on protecting the OS kernel integrity using virtualization technology. Secvisor [5] relies on memory virtualization to ensure that only verified code can execute in kernel mode so as to protect the kernel integrity against code injection attacks. Min Xu et al. propose a VMM-based architecture to detect and prevent all kernel integrity violations and use a usage control model for policy specification and enforcement to ensure the kernel integrity of guest OS [12].

Some research works aim at detecting malware based on certain symptoms exhibited by malware infection. Antfarm [9] enables VMM to implicitly discover and exploit the process information of the guest OS, thus can be used to detect hidden malware process. Lycosid [18] also implicitly obtains information about guest OS, and uses cross-view validation method to detect maliciously hidden OS processes based on these information. VMwatcher [11] proposes a comparison-based stealthy malware detection mechanism, which is similar to the mechanism in Lycosid.

The work that is closest to ours is [16], which also employs a VMM-based approach to intercept system calls for system monitoring. However, it is implemented in Xen hypervisor, which needs to modify guest kernel, so it violates our second design goal.

VI. CONCLUSIONS

In this paper, we present the design and implementation of VSyscall, a VMM-based security system which intercepts system calls outside of the guest OS. A system call correlating method is proposed to establish the relations between system calls. The method is also used by VSyscall to monitor the behavior of the program in Guest OS. VSyscall adopts full virtualization technologies, and makes no modification to the guest OS, so can support varies OS including commodity OS and legacy systems. We have implemented and evaluated VSyscall in Qemu and KVM,
and our experiments indicate that it is effective and only incurs a small amount of runtime overhead.

Our ongoing work is to enrich our current implementation to support other popular guest operating systems, such as windows etc.

ACKNOWLEDGMENT

This work is partially supported by grants from China 973 Fundamental R&D Program (No. 2005CB321803), National Natural Science Foundation of China (Project No. 60703056) and China 863 High-tech Program (Project No. 2008AA01Z118). We would also like to thank members from Network Computing Research team in Institute of Advanced Computing Technology of Beihang University for their hard work.

REFERENCES