Subverting System Authentication With Context-Aware, Reactive Virtual Machine Introspection

Yangchun Fu, Zhiqiang Lin, Kevin W. Hamlen
Department of Computer Science, The University of Texas at Dallas
800 W. Campbell Rd, Richardson, TX, 75080
{yangchun.fu, zhiqiang.lin, hamlen}@utdallas.edu

ABSTRACT

Recent advances in bridging the semantic gap between virtual machines (VMs) and their guest processes have a dark side: They can be abused to subvert and compromise VM file system images and process images. To demonstrate this alarming capability, a context-aware, reactive VM Introspection (VMI) instrument is presented and leveraged to automatically break the authentication mechanisms of both Linux and Windows operating systems. By bridging the semantic gap, the attack is able to automatically identify critical decision points where authentication succeeds or fails at the binary level. It can then leverage the VMI to transparently corrupt the control-flow or data-flow of the victim OS at that point, resulting in successful authentication without any password-guessing or encryption-cracking. The approach is highly flexible (threatening a broad class of authentication implementations), practical (realizable against real-world OSes and VM images), and useful for both malicious attacks and forensics analysis of virtualized systems and software.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software*

General Terms

Security

Keywords

Authentication; Reverse Engineering; Virtual Machine Introspection

1. INTRODUCTION

Virtualization of traditionally physical machines and hardware is being widely adopted as a means of cutting costs, improving portability, and easing maintainability of computer systems. For security, virtualized systems are typically represented as encrypted file system and memory images that are only accessible by loading the image into a compatible VM for execution, and passing an authentication check (e.g., login password) posed by the interpreted guest operating system (OS). To prevent subversion by an attacker, the authentication mechanism is often protected by an array of anti-debugging logic, cryptographic security, code obfuscation, and self-checking, to thwart efforts from reverse-engineering and credential-theft.

While these protections are viewed by many experts as reasonably secure against typical, low-resource attackers equipped with standard hardware, we observe that the rise of virtualization has quietly undermined several assumptions foundational to this view. Specifically, nearly all standard OS authentication defenses implicitly assume that the *hardware* on which the OS is running is non-malicious and immutable. For example, the OS's anti-debugging logic assumes that the machine instructions it uses to detect rival processes have the semantics defined by the CPU architecture, and that those semantics do not change from one instruction to the next. Unfortunately, this assumption can be potentially violated by a malicious VM, which effectively allows even a low-resource attacker to rapidly implement virtual, custom "hardware" with arbitrary semantics at low cost.

Even though the threat of malicious VMs is generally known (e.g., Bluepill [37] and SubVert [26]), the high danger of this situation has been significantly underestimated in our opinion, due in part to the well-known *semantic gap* between VMs and the guest software they host. VMs do not have direct knowledge of high-level program abstractions, such as data structures, functions, or algorithms; they operate at the level of registers, bytes, and instructions. This raises a veil of obscurity that seems to make authentication-tampering through a VM even more difficult than tampering at the traditional process level. We believe this apparent obstacle is why no previous work to our knowledge has actually carried out precision tampering of general authenticators (e.g., `login`) through malicious VMs.

In this paper we show that recent advances toward bridging the semantic gap [5, 9, 11, 33, 34] have made such VM-based attacks much more feasible. To demonstrate, we present the design, implementation, and evaluation of a malicious VM Monitor (VMM) that employs a novel VMI [13] to identify and subvert security-critical instructions in virtualized authentication implementations. Our attacks succeed without any knowledge of program-level abstractions or source codes; thus, concealing or obfuscating sources is not an effective defense.

Our experiments showcase the success of our approach against authentication protections, which are the first line of defense for many computer systems. In general, authentication can be abstracted as a boolean conditional test $f(x) = c$, where x could be a password, challenge response (e.g., CAPTCHA), biometric (e.g., fingerprint), or other input (e.g., RFID key) provided by the user; and f is a (one-way) validation function that transforms x into an output comparable

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ACSAC '13 Dec. 9-13, 2013, New Orleans, Louisiana USA
Copyright 2013 ACM 978-1-4503-2015-3/13/12:.$15.00.
http://dx.doi.org/10.1145/2523649.2523664.

to correct digest c. Guessing x by brute force is intractable when the domain of f is large, and learning x from c or finding a weakness in f is difficult if f is well-protected. Instead, we observe that an attacker with a reactive VMI can relatively easily find and subvert the comparison operation ($=$) or c directly. By flipping the comparison outcome or replacing c with $f(y)$ for some known y, the attacker gains access without knowing x.

As a proof-of-concept, we present a new technique to break system authentication by designing a malicious VMM built atop a VMI [13] that pulls guest-OS state from the outside VMM and enriches the VMI with *context-aware*, *reactive* capability. That is, unlike traditional VMI, our VMI requires access to only a small subset of the guest-OS states, but offers write-access to states in addition to read-access. Through step-by-step design, implementation, and evaluation, we show that such a VMI capability can automatically intercept and tamper with any of the execution states of the authentication process (e.g., `login`, `sshd`, `vsftpd`, or `telnetd` in Linux/UNIX, and `winlogon` in Microsoft Windows) running inside the guest-OS, and thereby enter the target system without knowing any password.

While the virtualzation layer can theoretically intercept and tamper with any guest program states, our main technical challenge is to identify the exact execution context to modify, and modify it appropriately. That is, *we must identify and corrupt the precise execution state at the instant of authentication failure, and infer a state change that makes it succeed.* However, there are no abstractions (e.g., process descriptors, file objects, or variables) at the VM layer. This grand obstacle explains why, to the best of our knowledge, there have been no such VMI attacks in the wild previously. We show how semantic gap-bridging technologies applied using our VMI can overcome this obstacle.

Our VMI attack is extremely dangerous and practical. We demonstrate that it can automatically break the authentication and enter the target system without any password. This implies that malicious public or private cloud providers can harvest sensitive information from user-supplied VM images even if the images might have been protected by full disk encryption (FDE), since malicious cloud providers can use our attack to authenticate while the VM is running. Moreover, the current trend of virtualizing physical machines [42] dramatically increases exposure to our attack, since virtual system images can be loaded into our malicious VMM to bypass their authentication.

The attack is also extremely flexible. With state-of-the-art binary code analysis techniques (e.g., [20]), attackers can dynamically patch any instructions or execution states of victim applications (not just the authentication process) to bypass other security measures, such as license checks and commercial OS activation checks. It is also potentially useful for benign purposes, such as to facilitate law-enforcement access to authentication-protected criminal laptops (by first converting it to a VM and then executing atop our VMI).

In summary, this paper makes the following contributions.

- We present a new technique to break authentication using VMI, and show that it is extremely powerful, allowing attackers to trivially enter victim computers without a password.
- We devise a novel enabling technique—context-aware, reactive introspection—that enriches traditional read-only VMI with guest CPU register- and memory-write capability, and we present a detailed design, implementation, and evaluation.
- We observe that our VMI attack is very difficult to prevent for any system amenable to virtualization. For protection of laptops, we advocate FDE with *pre-boot authentication*.

2. BACKGROUND

We begin by examining various possible approaches to subverting system authentications in §2.1, and then define our threat model and assumptions in §2.2.

2.1 Possible Approaches

There are four obvious approaches to subverting authentication test $f(x) = c$: (1) computing $x = f^{-1}(c)$, (2) tampering with f, (3) tampering with equals ($=$), or (4) replacing c with $f(y)$ for some known y. Computing $x = f^{-1}(c)$ through brute force (e.g., dictionary attack) is historically the most common, but requires enormous computing resources if the domain of f is large and x is non-trivial. Tampering with f is frustrated by its usually (intentionally) complex implementation, which include numerous cryptographic layers, anti-debugging, and self-checks.

In contrast, the implementation of equals is relatively simple; it usually manifests as a single conditional jump instruction (`je`/`jne` on x86) hidden somewhere in the binary. If equals is later followed by a double-check, such as a control-transfer of the form `jmp` $g(x, c)$ where $g(x, c)$ yields some correct destination d if and only if $f(x) = c$, then this extra check is similarly subvertable once d is known (e.g., by observing the behavior of a program copy where legitimate authentication is possible).

Likewise, replacing c with $f(y)$ is often an attractive and easy alternative approach. In this scenario, the attacker uses the malicious VMI to effect a man-in-the-middle (MitM) attack that replaces the resource containing c (e.g., the password file or its in-memory image) with a false one that encodes attacker-owned credential y. The attacker can then successfully authenticate using y instead of x.

There are three layers at which one might subvert equals or c in modern computing infrastructures:

Hardware-layer. One approach is to craft malicious hardware that corrupts the critical comparison instruction or credential digest but leaves all other instruction semantics unaffected. This is conceptually appealing since faithfully preserving all other instructions makes it impossible for any software at the upper layers to inspect and detect the malicious environment. However, design and implementation of trojaned hardware that nonetheless meets the complex operational demands of standard hardware requires resources beyond the capabilities of most typical attackers. Hardware companies with this capability are not motivated to carry out such attacks since doing so potentially harms their reputation with customers. Thus, aside from malicious hardware crafted by determined, capable adversaries in the defense and research communities (e.g., [4, 7, 27, 38]), such attacks are relatively rare to our knowledge.

Software-layer, inside OS. In contrast to hardware, the software layer has long been an affordable target for all levels of attacks (from kernel rootkits to backdoors, trojans, and viruses). More specifically, software attackers commonly attempt to break authentication $f(x) = c$ by introducing a backdoor that circumvents f, patching the equality check after f, or forging data c. However, for these attacks to succeed, there are two conditions: (1) we must be able to mount the disk of the target system and traverse its data files to locate f and c, and (2) the target system must not contain any integrity checks (e.g., Tripwire [25]) to verify the integrity of f and c as well as other critical components.

With the increasing privacy and data breach concern from endusers, most modern OSes satisfy neither of these conditions. Disk images are usually encrypted (especially those in IaaS clouds and on laptops protected by products like BitLocker [31]), and OSes implement self-checking and anti-debugging technologies that detect corruption at the software layer. It is thus difficult to precisely locate

f, c, or the conditional branch in the target disk, and corrupting them raises OS integrity alarms that can render the system unusable after rebooting. In other words, directly tampering at the OS and user-level can lead to easy detection by the target computer.

Therefore, since hardware-layer attacks are hard to implement and software-layer inside OS attacks are easy to detect, this leads our investigation to the virtualization layer that is outside OS.

Software-layer, outside OS. The rise of virtualization technologies as the underpinnings for today's cloud computing and data centers has made it truly practical for low-resource attackers to create virtual "hardware" using software, with arbitrary semantics on demand at low costs. We show that with only minor additions in the form of context-sensitive, reactive VMI, these technologies facilitate new, cheap attacks that are undetectable to typical defenses employed by state-of-the-art guest OSes. These attacks come at almost zero cost for attackers—with only a few thousands lines of code (LOC) added to an existing virtualization platform, one can implement Blue Pill-like attacks [37] that are very challenging, if not impossible, to detect above the virtualization layer, because they run entirely beneath the OS and have complete control of the guest system.

2.2 Threat Models, Scope and Assumptions

Threat Models. Our goal is to break the target's authentication defense using a stealthy dynamic method from the virtualization-layer (without being detected by the in-guest security software). We assume unfettered access by the attacker to a VM image of the target system. For example, if the target is a physical machine to which the attacker has physical access, the attacker can virtualize the target to a VM image using standard tools (e.g., [42]) or custom tools to satisfy this assumption.

If the target is encrypted with FDE, our goal is to view the encrypted data by breaking the authentication. Physical machines with pre-boot FDE authentication are immune to our attack since to really decrypt the data, we have to provide the correct key. This is different compared to authentication subversion. But if the FDE machines are protected by post-boot authentication, they are vulnerable.

Scope and Assumptions. In this paper, we focus on subverting systems atop x86 architectures. Since the attacker owns the virtualization, none of the virtualization-based defenses in the recent literature (e.g., [14, 22, 46]) are applicable to detecting our attack. However, we pessimistically assume the presence of powerful in-guest security software, such as rootkit detectors and integrity checkers. Our goal is to undermine these by tampering with their state (similar to our authentication tampering).

We also assume that attackers have access to standard binary reverse engineering tools (e.g., [6, 20, 28]). In this paper we do not cover or improve any of these existing techniques; we apply them to gain instruction-level knowledge of the victim binary code.

Attackers could be malicious cloud providers, benign providers that have been compromised (perhaps by insider threats), individual users such as script kiddies and criminals, or even law-enforcement officials attempting to penetrate criminal-owned computers.

3. ATTACK OVERVIEW

3.1 Working Example

We illustrate our approach using a working example that targets the UNIX/Linux `login` program (from `shadow-4.1.4.2`) as the victim software. Our goal is to dynamically tamper with the program's execution state to gain access to the system without knowing any password.

```
if (pw_auth (user_passwd, username, reason, (char *) 0) == 0) {
 804a868:    a1 0c 62 05 08            mov    0x805620c,%eax
 804a86d:    c7 44 24 0c 00 00 00      movl   $0x0,0xc(%esp)
 804a874:    00
 804a875:    89 3c 24                  mov    %edi,(%esp)
 804a878:    89 44 24 08               mov    %eax,0x8(%esp)
 804a87c:    a1 48 65 05 08            mov    0x8056548,%eax
 804a881:    89 44 24 04               mov    %eax,0x4(%esp)
 804a885:    e8 86 87 00 00            call   8053010<pw_auth>
 804a88a:    85 c0                     test   %eax,%eax
 804a88c:    0f 84 6d fd ff ff         je     804a5ff<main+0x64f>
            goto auth_ok;
}
```

Figure 1: Binary Code Snippet of the `login` Program.

```
  1 execve("/bin/login", ["login"], [/* 16 vars */]) = 0
  2 uname({sys="Linux", node="ubuntu", ...}) = 0
...
409 open("/etc/passwd", O_RDONLY)            = 4
410 fcntl64(4, F_GETFD)                      = 0
411 fcntl64(4, F_SETFD, FD_CLOEXEC)          = 0
412 _llseek(4, 0, [0], SEEK_CUR)             = 0
413 fstat64(4, {st_mode=S_IFREG|0644, st_size=952, ...}) = 0
414 mmap2(NULL, 952, PROT_READ, MAP_SHARED, 4, 0) = 0x4021a000
415 _llseek(4, 952, [952], SEEK_SET)         = 0
416 munmap(0x4021a000, 952)                  = 0
417 close(4)                                 = 0
418 open("/etc/shadow", O_RDONLY)            = 4
419 fcntl64(4, F_GETFD)                      = 0
420 fcntl64(4, F_SETFD, FD_CLOEXEC)          = 0
421 _llseek(4, 0, [0], SEEK_CUR)             = 0
422 fstat64(4, {st_mode=S_IFREG|0640, st_size=657, ...}) = 0
423 mmap2(NULL, 657, PROT_READ, MAP_SHARED, 4, 0) = 0x4021a000
424 _llseek(4, 657, [657], SEEK_SET)         = 0
425 munmap(0x4021a000, 657)                  = 0
426 close(4)                                 = 0
...
```

Figure 2: System Call Trace Snippet of the `login` Program.

Figure 1 shows a static binary code snippet of `login`, which calls the `pw_auth` function for authentication. A system call (*syscall*) trace is presented in Fig. 2. It opens the "`/etc/passwd`" and "`/etc/shadow`" files and maps their file contents into memory (using the `mmap2` syscall).

3.2 Attack Method

In general, we can partition our attacks into two classes based on instrumentation granularity. The first class of attacks dynamically tampers with the instruction execution state (e.g., the instruction operand). We term this an INSTVMI attack, because the instrumentation is at the instruction level and it relies on instruction translation or emulation-based software virtualization. INSTVMI usually runs in emulation-based software virtualization, such as QEMU. (INSTVMI can also be implemented using a single step execution mode in hardware virtualization, but we eschew this option as more difficult.) The second class of attacks tampers with the syscall execution state (e.g., the syscall arguments and return values). We term these SYSVMI attacks, because the instrumentation is at the syscall level and can be implemented using hardware virtualization, such as Xen/KVM. Thus, we can use the following strategies:

Instruction Execution Tampering. Since the `login` authentication occurs in function `pw_auth`, we can choose from a number of candidate instructions (boxed in Fig. 1) as well as their data dependencies for the tampering, resulting in various approaches:

- **Tampering with Instruction Opcodes**. Machine instruction opcodes can be corrupted to change the program's semantics. For instance, the VMM can locate virtual address 804a885 of the `login` process right after it is loaded into memory, and directly change the machine code bytes e8 86 87 00 00 (which encode the `call 8053010` instruction), to b8 00 00 00 00

(mov $0, %eax); or it can locate virtual address 804a88c and change opcode 0f 84 (je) to 0f 85 (jne). Note that at the VMM layer there is no read-only protection of the instruction code, leaving the VMM free to corrupt arbitrary bytes.

- **Tampering with Instruction Operands**. The VMM can also corrupt instruction operands, such as zeroing the return value (in eax) of the function call to pw_auth at 804a885. It can also modify the processor status flags (EFLAGS) resulting from test instruction "test %eax, %eax" at 804a88a. For this type of tampering, we must dynamically intercept the instruction. Also, while instruction opcode tampering can be theoretically detected by periodic (and often computationally expensive) memory integrity checking, none of the existing techniques can detect register value tampering.

These tampering techniques from the virtualization-layer facilitate successful authentication irrespective of the password entered, affording unrestricted access to the target system. While opcode tampering can be implemented without instruction level monitoring (e.g., by changing the opcode right after the code is loaded at the syscall event level), operand tampering must precisely capture the execution context (e.g., when the program counter reaches a specific virtual address) at the instruction level.

Syscall Execution Tampering. Aside from instruction-level tampering, we can also launch syscall-level attacks to tamper with data that is directly or indirectly related to $f(x) = c$. For example, we can forge c with a hash value generated from our own password. Notice that for this particular login example, as shown in Fig. 2, the system opens a disk file /etc/shadow that stores the password hash values, and maps (mmap2) the file contents to memory. As such, we present two strategies:

- **Tampering with Disk-IO Syscalls**. Even though we do not know the password in /etc/shadow, we can replace this file with a file we provide from the virtualization-layer. This is actually a MitM attack that redirects the file-open syscall to an attacker-controlled password file.

- **Tampering with Memory-Map Syscalls**. If it is difficult to provide an attacker-controlled password file at the virtualization layer (e.g., due to the semantic gap), there is an even simpler attack: We can attack the mmap2 syscall by replacing the memory contents mapped by this syscall (immediately after it finishes) with the forged password hash values.

Through syscall execution tampering, the login process consumes security-critical data, such as the password hash values, that are controlled by attackers. Full access is thereby granted by entering a known password. The syscall execution tampering approach is more appealing than instruction tampering, since it is more transparent to the victim code, and does not require a detailed instruction level understanding of the victim application. Our experiments demonstrate that we can transparently break the authentication process of sshd, vsftpd, telnetd, and winlogon without any reverse engineering of their binary code by using syscall execution tampering.

3.3 Overview

To realize the above attacks, we must bridge the semantic gap to precisely identify the target concrete execution contexts at the VMM layer. To do so, we present a *context-aware, reactive* VMI as a foundation for our attack. It has the following features:

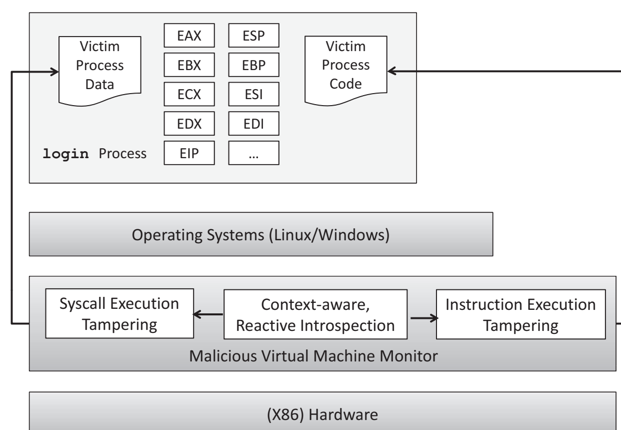

Figure 3: Architecture Overview of Our Attack.

- **Introspection**. Our approach is introspective [13], since it runs outside the guest-OS and interprets certain guest events, such as particular syscall executions (e.g., the arguments to syscall open and the return value of mmap2).

- **Reactive**. Unlike the traditional *passive, read-only* introspection techniques (e.g., [9, 11, 13, 34]), our approach is *reactive*. That is, attacker-defined tampering actions are triggered by certain attacker-defined execution contexts.

- **Context-aware**. The contextual criteria that trigger attacker actions can range over process level, syscall level, call stack level, and instruction level properties, depending on the attack strategies.

4. DESIGN AND IMPLEMENTATION

To show the generality of our techniques, we have designed two sets of attacks based on our context-aware, reactive VMI and different types of virtualization: (1) SysVmi, which uses hardware virtualization to perform syscall tampering; and (2) InstVmi, which uses software virtualization to perform instruction level tampering. In the following, we present the step-by-step design and implementation of these techniques. We first present SysVmi in §4.1 since it is the easier and more appealing of the two, followed by InstVmi in §4.2.

4.1 SysVmi

The essence of our attack is to tamper with the program state at precise moments guided by execution context details identified at the virtualization layer. In general, an execution context can belong to the following categories (from coarse- to fine-grained):

- **C1**: a particular process execution;
- **C2**: a particular syscall in **C1**;
- **C3**: a particular instruction in **C1**;
- **C4**: a particular instruction (**C3**) under a particular call stack.

However, at the virtualization-layer, there is almost no abstraction. Thus, the following sections examine how to bridge the semantic gap to identify coarse-grained execution contexts **C1–2** (see §4.1.1) and fine-grained contexts **C3–4** (see §4.2). We then present the design and implementation of SysVmi in §4.1.2 and §4.1.3, respectively.

4.1.1 Coarse-Grained Context Awareness

Process execution context identification (C1). Nearly all modern OSes for x86 architectures use paging to support isolated, private virtual address spaces for processes. Specifically, each process has a private page directory (`pgd`) to which control register CR3 typically points; the value of CR3 can hence be used to differentiate processes. To the best of our knowledge, nearly all x86 introspection techniques use CR3 to isolate the process execution context (cf., [21, 22]).

However, the value of CR3 does not directly reveal process identities; for example, it does not reveal the process's name. Process identities are needed for our attack to precisely and surgically corrupt only the security-relevant code and data without altering the behavior of the rest of the system. One way to derive process identities from CR3 is to traverse the guest OS data structures where that data is stored (e.g., `task_struct` in Linux and `EPROCESS` in Windows). However, the resulting implementation is specific to fine details of the guest OS version, and therefore does not generalize well.

We therefore developed a kernel-independent approach that combines binary code fingerprinting and the CR3 identification from Antfarm [21] to uniquely identify victim processes from the virtualization layer. Rather than using process names (which could be modified by cloud users trying to evade our attack), we hash (MD5) the code page in which each process's main entry point resides, and then compare the hash values at run-time to identify the target process. Process entry points can be acquired by disassembling the binary code, and are also divulged when pushed onto the stack by `__libc_start_main`, whose symbol is always present in dynamic linked binaries. They are difficult to obfuscate and were not obfuscated in any of the binaries we studied, since they must be divulged to the guest OS's process loader via standard interfaces in order to load the process.

There are more sophisticated binary code fingerprinting techniques, such as byte-signatures [24], string-signatures [16], and semantic-aware signature [45], that would also work. However, our simple hash-based signature sufficed for all our experiments. In addition, our experiments only required process-level context tracking, not thread-level tracking. Thread tracking at the hypervisor layer can also be realized (e.g., [12]) if needed.

Syscall execution context identification (C2). Syscalls are exported OS services with standard interfaces. In x86, syscalls are implemented via unique instruction pairs. The Linux kernel uses `int0x80/iret` and `sysenter/sysexit` pairs, and Microsoft Windows uses `int 0x2e/iret` and `sysenter/sysexit` pairs. The callee of the syscall is indexed by register `eax` when invoked.

With the instruction translation-based virtualization presented in §4.2, it is trivial to identify the specific syscall execution context. With hardware-assisted virtualization (e.g., Xen and KVM), we must rely on other hardware mechanisms to intercept the execution of the syscall instructions. Ether [8], built atop the Xen hypervisor, leverages page fault exceptions to capture syscall enter and syscall return points. Nitro [35], based on the KVM hypervisor, leverages invalid opcode exceptions to intercept syscalls. In addition, hardware breakpoints can also be used [43]. Here, we adopt the page fault exception approach of Ether and synthesize a page fault at each syscall entry and syscall return point:

- **Syscall-Entry**. We copy the value of the original `0x80` (`0x2e` for Windows) IDT entry, and substitute it with a unique invalid memory address. A similar method applies to `sysenter`-based syscalls using the `SYSENTER_EIP_MSR` register. All syscall invocations therefore result in a page fault exception, where we can detect the syscall entry point by inspecting the invalid address.
- **Syscall-Return**. To intercept syscall returns, we set the page containing the return address as inaccessible in the shadow page table. This return address page is identified at the syscall entry point.

4.1.2 Detailed Design

The design of SysVMI is based on hardware virtualization. In particular, it achieves coarse-grained execution context identification and performs syscall-level *reactive* tampering. In the following, we describe the design of each specific reactive tampering attack.

A1. Tampering with Instruction Code. Right after a victim process is loaded—for instance, at the syscall exit point after executing the `execve` call in line 1 of Fig. 2—we fetch the target page in which the victim PC resides (e.g., `804a885`, `804a88c`), and directly tamper with its machine code. This attack is general, but requires a detailed understanding (and binary analysis) of the victim application.

A2. Tampering with Syscall Arguments and Return Values. Rootkits often tamper with syscall arguments and return values (e.g., to hide the presence of a malicious process). Since we have the capability to intercept the syscall entry and syscall return, it is trivial to tamper with any of the arguments and return values of interest to attackers. Although our `login` process attack does not solely involve such tampering, it is needed in many other attack scenarios. For example, it is needed by certain attacks that corrupt the system log by disabling syscalls that write to certain files.

A3. Tampering with Syscall-produced Data. Whenever there is a data dependency between program-consumed and syscall-produced data, there is an opportunity for a spoofing MitM attack. The syscall return value tampering in **A2** is a special case of this more general class of attacks. In the case of our `login` attack, right after the execution of `mmap2` when mapping `/etc/shadow` to memory, we can replace the memory chunk with the file content from an attacker-controlled file. Such spoofing is very difficult to prevent or detect, and does not require any sophisticated reverse engineering of the binary code.

A4. Using IO Virtualization. If attack **A3** requires syscall level knowledge, such as the semantics of `mmap2`, then **A4** lifts this requirement by using IO virtualization [1, 40]. More specifically, since `login` processes eventually open the `/etc/shadow` file and read it through disk IO, we can intercept the data transmission from disk IO to memory to successfully spoof the file.

To avoid the need for file name abstractions, which are not readily available at the IO virtualization layer, we can leverage information available from the trace in Fig. 2 to identify the victim IO transmission by its content. For example, the `/etc/shadow` file content can be identified by using parsing to match its general syntax, or by fingerprinting its exact contents, and waiting for a matching IO transfer. Depending on the attack scenario, the syntax or fingerprint can be revised, so we believe this methodology also generalizes to many attacks.

4.1.3 Implementation

We have implemented SysVMI atop a recent Xen hypervisor (Xen-4.12), and supported attack methodologies **A1**–**4** with 1,895 LOC in total. Table 1 reports a size breakdown of our SysVMI implementation. The size of our context-aware reactive VMI is presented in column 2, and each specific attack is presented from column **A1**–**4**. The table shows that once the introspection foundation of **C1**–**2** is established, the implementation of attacks **A1**–**4** is relatively trivial. This indicates that our VMI approach is easily extensible to many attacks, with new tampering attacks being easily implementable based on attacker needs.

VMM	C1–2	A1	A2	A3	A4	Total
Xen-4.12	1,748	17	10	75	45	1,895

Table 1: Code Size (LOC) of Our SysVmi Implementation

4.2 InstVmi

Next, we present the detailed design and implementation of InstVmi. Since a VMM based on software virtualization emulates hardware entirely through binary code translation, it has complete control of the system, including all the capabilities of SysVmi (e.g., introspection levels **C1**–**2** and attack methods **A1**–**4**). To avoid redundancy, we therefore limit our discussion in this section to the new introspection opportunities and tampering attacks afforded by fine-grained execution context identification.

4.2.1 *Fine-Grained Context Awareness*

Instruction execution context identification (C3). We use the program counter (PC) to identify the specific instruction execution context under **C1**. This is very trivial, since we control the binary code translation, and we can therefore instrument it to tamper with the execution precisely when the PC visits certain addresses of our interest.

Call-stack context identification (C4). In addition to the PC, the call stack and the return addresses it contains can also be used to more precisely describe the execution context, as demonstrated in VtPath [10]. To collect the call-stack, we instrument `call` and `ret` instructions. Whenever a `call` gets executed in our monitored process (**C1**), we push the return address and the current `esp` value onto a private shadow stack, and whenever a `ret` gets executed for the monitored process, we pop from our shadow stack until it matches the right `esp`. Tracking `esp` is necessary to identify and avoid mismatched `call`/`ret` pairs.

4.2.2 *Detailed Design*

The facility to identify these fine-grained execution contexts offers attackers extremely powerful tampering attacks. Since we are able to track the execution of each instruction, we can achieve coarse-grained syscall context identification (**C1**–**2**) by simply intercepting `int 0x80`, `sysenter`, and `sysexit` instructions. It is very simple to identify these syscall contexts, as illustrated by past work [9, 11]. In addition to attacks **A1**–**4**, which can be realized by SysVmi, the following new attacks become possible with InstVmi:

A5. Tampering with Instruction Code at PC Level. While **A1** is already able to tamper with the instruction code, its granularity is at syscall level and lacks flexibility. For more precise tampering, SysVmi can tamper with the machine code only when the instruction at a particular virtual address is about to execute.

A6. Tampering with Instruction Operand. The most intuitive and fine-grained tampering is to forge an instruction's register or memory operand. Our framework facilitates surgical alteration of CPU registers without corrupting the rest of the execution. Effectively using this capability requires a detailed reverse engineering of the binary code, in order to single out the exact victim operand and PC address.

A7. Tampering with Function Call Arguments and Return Values. In **A2**, we are only able to tamper with the syscall related arguments and return values. In **A7**, we can now tamper with any function call arguments, including function calls in user space or library space, as long as attackers specify which function to corrupt (via a configuration interface in our design).

VMM	C1–2	A1	A2	A3	A4	Total
QEMU-1.01	1,250	22	30	38	48	1,388

Table 2: Code Size (LOC) of the InstVmi$_a$ Implementation

VMM	C1–4	A5	A6	A7	Total
QEMU-1.01	3,513	35	34	25	3,607

Table 3: Code Size (LOC) of the InstVmi$_b$ Implementation

4.2.3 *Implementation*

We have implemented two kinds of InstVmi: InstVmi$_a$ ports the SysVmi implementation (**C1**–**2** and **A1**–**4**) to QEMU-1.01, and InstVmi$_b$ implements the new attacks unique to software virtualization (**A5**–**7**) with fine-grained execution context identification (**C3**–**4**). A size breakdown of both are presented in Table 2 and Table 3, respectively.

A comparison of Tables 1 and 2 shows that identifying process and syscall level context is easier using approach InstVmi$_a$, whose total implementation is about 500 LOC smaller. Adding support for **C3**–**4** and **A5**–**7** requires InstVmi$_b$ to dynamically instrument each instruction to check the execution context, resulting in a larger 3,513 LOC implementation for the context identification in Table 3.

5. EVALUATION

This section presents our experimental results. We first describe the effectiveness of each of our attacks (**A1**–**7**) in §5.1, and then study the performance overhead in §5.2. Next, §5.3 evaluates the generality of our attacks with respect to different software applications running on different platforms. Our testing system is a Dell workstation with Intel Core2 Quad Processor and 24GB of RAM. The guest-OSes are Linux-2.6.32 and Windows XP (SP2), and the host OS is Linux-3.0.1 for InstVmi. The SysVmi hypervisor is Xen-4.12.

5.1 Effectiveness

In addition to UNIX `login` (our running example), our authentication software victims include `sshd`, `vsftpd`, and `telnetd`, which all have an authentication component. We tested whether each of our attacks is effective against each program.

`login`. Using SysVmi, we applied attacks **A1**–**4**, except with **A2** and **A3** combined to implement the concrete attack (i.e., enabling a root user entering the system without knowing any password). The general interface for each attack is adequate to launch the attack without developing any new code; we simply configure the concrete address and instruction code to modify (**A1**), the syscalls and the spoofed content (**A2** and **A3**), or the file pattern of the spoofing hijacking (**A4**).

Likewise, InstVmi$_a$ and InstVmi$_b$ are similarly configurable (without any code modification) to successfully launch attacks **A1**–**4** to subvert the `login` process. For the new unique attacks **A5**–**7** available with InstVmi$_b$, we configure **A5** to tamper with the machine code at `0x804a885` without actually calling the `pw_auth`, we configure **A6** to tamper with the EFLAGS register at instruction `0x804a88c`, and we configure **A7** to tamper with the return value of function call `pw_auth` at `0x804a88a`. All of these attacks succeed.

`sshd`. The `sshd` authentication program (from `openssh-5.8`), whose binary code is excerpted in Fig. 4, offers at least two tampering opportunities for **A1**: change the two arguments to function `strcmp` (at `0x80517e2` and `0x80517e6`) to be identical, or change

```
encrypted_password = xcrypt(password,
    (pw_password[0] && pw_password[1]) ? pw_password : "xx");
return (strcmp(encrypted_password, pw_password) == 0);

80517da:    89 34 24                mov     %esi,(%esp)
80517dd:    e8 4e 3b 04 00          call    8095330 <xcrypt>
80517e2:    89 5c 24 04             mov     %ebx,0x4(%esp)
80517e6:    89 04 24                mov     %eax,(%esp)
80517e9:    e8 ae b3 ff ff          call    804cb9c <strcmp@plt>
80517ee:    85 c0                   test    %eax,%eax
80517f0:    0f 94 c0                sete    %al
80517f3:    0f b6 c0                movzbl  %al,%eax
80517f6:    8b 55 f4                mov     -0xc(%ebp),%edx
80517f9:    65 33 15 14 00 00 00    xor     %gs:0x14,%edx
8051800:    75 39                   jne     805183b <sys_auth_passwd+0x9b>
```

Figure 4: Code snippet of the targeted `sshd` program.

```
p_crypted = crypt(str_getbuf(p_pass_str), p_pwd->pw_passwd);
if (!vsf_sysutil_strcmp(p_crypted, p_pwd->pw_passwd))
{
  return 1;
}

1b465:    e8 56 8f fe ff    call    43c0 <crypt@plt>
1b46a:    89 45 e0          mov     %eax,-0x20(%ebp)
1b46d:    8b 45 ec          mov     -0x14(%ebp),%eax
1b470:    8b 40 04          mov     0x4(%eax),%eax
1b473:    89 44 24 04       mov     %eax,0x4(%esp)
1b477:    8b 45 e0          mov     -0x20(%ebp),%eax
1b47a:    89 04 24          mov     %eax,(%esp)
1b47d:    e8 4e d2 ff ff    call    186d0 <vsf_sysutil_strcmp>
1b482:    85 c0             test    %eax,%eax
1b484:    75 07             jne     1b48d <vsf_sysdep_check_auth+0x161>
```

Figure 5: Code snippet of the targeted `vsftpd` program.

the `jne` to `je` (at `0x8051800`). For the combined **A2–3** attack and attack **A4**, the generality of these two spoofing attacks facilitate replacing the old `/etc/shadow` with an attacker-controlled file to bypass the password check. By directly applying these SYSVMI attacks to INSTVMI$_a$, we successfully logged into the system.

For the fine-grained tampering available via INSTVMI$_b$, we tamper with the instruction code at `0x8051800` (for **A5**), tamper with the instruction operand at `0x80517ee` (`test eax, eax` for **A6**), and function argument of `strcmp` at `0x80517e2` (changing the `eax` to `ebx` for **A7**). This is just one of many available attack vectors. For example, we could have tampered with the EFLAGS at `0x8051800` (**A6**), and tampered with the argument at `0x80517e6` (e.g., changing the `ebx` to `eax` for **A7**).

`vsftpd`. We attacked an ftp daemon from `vsftpd-3.0.0` to test whether we are able to break its authentication. The disassembly fragment in Fig. 5 shows that it has an authentication pattern similar to that of `sshd`. Using SYSVMI **A1** and INSTVMI$_b$ **A5**, we change the opcode of `jne` to `je` at `0x1b484`. The syscall **A2–3** and IO tampering **A4** attacks succeed with the same configuration as used in the `login` and `sshd` attacks.

For other INSTVMI attacks, we again observed many available attack options. For example, corrupting the operand of `test eax, eax` at `0x1b482` (**A6**), or the EFLAGS register at `0x1b484`, or the argument of `vsf_sysutil_strcmp` (**A7**), all result in successful compromises.

`telnetd`. Application `telnetd` (in `netkit-telnet-0.17`) internally uses the `login` process to perform the authentication. As such, all the attack methods in `login` are successful against `telnetd` without modification.

Summary. From these experiments, we confirm that attack methods **A2–4** in SYSVMI are all transparent to these victim programs. Each attack works against all the victims without requiring the attacker to extend or customize the implementation in any way. Other attacks, including **A1** and **A5–7** (in INSTVMI), require analysis of the victim binary code to configure which PC, instruction operand or opcode, or function calls to corrupt.

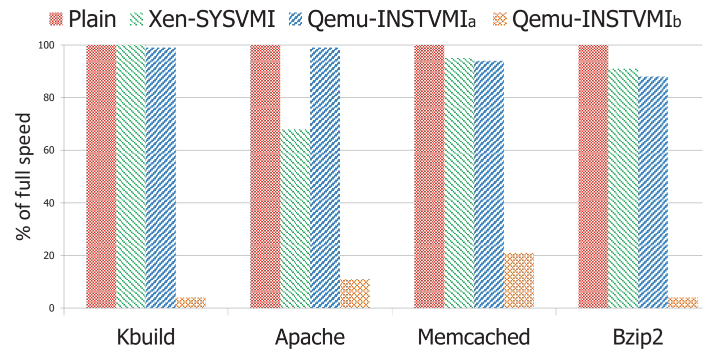

Figure 6: Macro-benchmark Evaluation of the Performance Overhead of Our VMI.

5.2 Performance Overhead

Tables 1–3 show that each attack (**A1–7**) has a very small implementation, which is only triggered at certain specific execution contexts. The majority of the performance overhead stems from the instrumentation of our context-aware, reactive VMI. To evaluate this overhead, we used standard system benchmarking programs to measure the runtime overhead at both the macro and micro level.

Macro Benchmarks. We used kbuild (which is CPU and disk intensive), ApacheBench-2.2.15 [2] (which is network I/O intensive), memcached-1.4.5 [30] (which is memory and I/O intensive), and bzip2 (which is CPU and disk intensive) to quantify the performance slowdown at a macro level.

For kbuild, we build a compact kernel by running the command "`make allnoconfig`", and record the time spent on the compilation of Linux kernel 2.6.32. For ApacheBench, we test the Apache server throughput with over 10,000 requests of a 4K-byte file in a Apache server. For memchached, we use a remote client to issue 1,000 write and read requests to a memchaed server. For bzip2, we decompress the official Linux 2.6.32 kernel source tarball and record the processing time.

We normalized the performance overhead of these benchmarks to each of our VMI implementations. The performance overhead is presented in Fig. 6. When there is more user level code computation, our system has a small overhead (as shown in kbuild and bzip2 case). For ApacheBench, because of the syscall overhead we introduced in SYSVMI, the network response time decreases. For memcached, both the Xen and QEMU implementations have less overhead because this benchmark is memory intensive. For our instruction-level VMI, as expected, it has huge performance overhead for all the benchmarks, since we instrumented each instruction execution.

Micro-benchmarks. To evaluate the primitive level performance slowdown, we use the standard micro-benchmark LMBench suites to estimate the VMI's impact on various OS operations. In particular, we focus on the overhead that instrumentation introduces to context switches, page faults, memory-related operations (such as memory map), and IO-related operations (such as TCP and disk files).

The results are presented in Fig. 7. Recall that for Xen, we use page faults to intercept syscalls, which leads to VMExits and VMEntries to resume the execution of the VM. This introduces some overhead, which is why the overheads for TCP/UDP and Mmap are high. Bcopy does not use syscalls, so it runs as fast as the original. Regular (non-introduced) page faults do not incur any measurable slowdown. Since QEMU already emulates each instruction, QEMU-INSTVMI$_a$ requires significantly less overhead

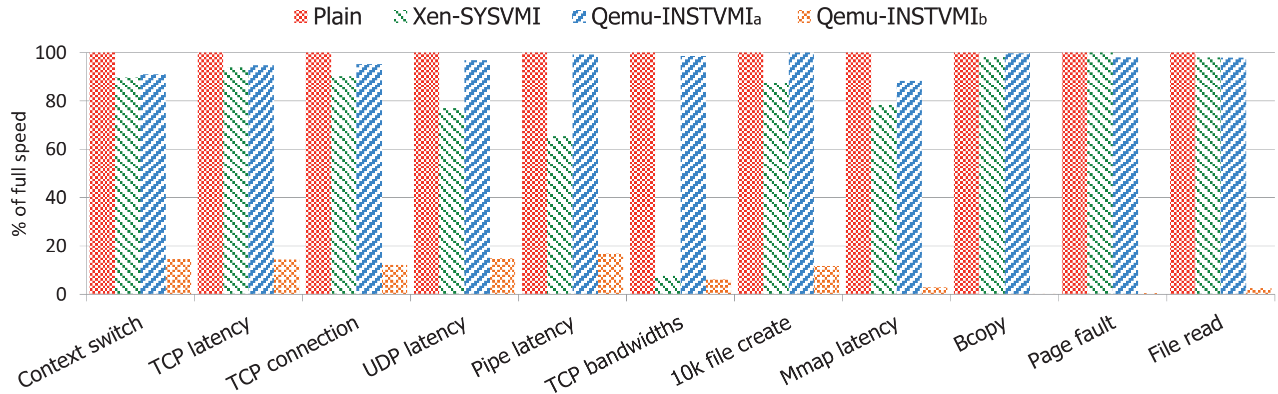

Figure 7: Micro-benchmark Evaluation of the Performance Overhead of Our VMI.

to capture each syscall; the only overhead comes from our syscall attack code. That is why QEMU-INSTVMI$_a$ compares favorably to Xen-SYSVMI. For QEMU-INSTVMI$_b$, since each instruction is monitored to determine the execution context, it has significantly worse performance.

Summary. From the performance overhead evaluation, we conclude that the slowdown at the process level is light if we use the SYSVMI attack. Therefore, a malicious cloud provider could possibly deploy this attack online (as a backdoor to access victim jobs, for instance). Even though instruction level INSTVMI has larger overhead, we believe attackers still have incentives to use it for such tasks as offline analysis, considering the huge benefits they might glean from successful attacks.

5.3 More Case Studies

We have demonstrated in the previous sections that we can successfully compromise Linux authentication programs. In this section, we demonstrate how only small changes to our attack code suffices to compromise Windows authentication. Specifically, we report two case studies: (1) tampering with a Windows `winlogon.exe` authentication, and (2) tampering with serial number validation in a Windows program. To avoid any copyright issues, we did not perform any reverse engineering of Windows `winlogon.exe`, and we targeted a generic serial checker binary program for demonstration purposes.

Tampering with `winlogon.exe`. Since legal concerns dissuade us from inspecting the binary code of `winlogon.exe`, we just use our high level knowledge of this program and employ our transparent SYSVMI attack. More specifically, we have a general understanding that `winlogon.exe` accesses the Windows password file `sam` (just as `login` accesses `/etc/shadow`). By intercepting the Windows file open system call, we can detect this file access. Windows loads `sam` automatically during system booting (this load is not requested by `winlogon.exe`), but this does not impair the attack. We use syscall tampering (**A3**) to inspect all file-loads. Once we detect that `sam` is loaded, we spoof the OS with a file we own. As such, when `winlogon.exe` checks the password in `sam`, we successfully log into the system.

To test whether our attack is feasible when the target disk is encrypted, we then installed a FDE system configured without preboot authentication in a Windows XP (SP2) VM. Next, we run the VM images in our malicious VMM. Unsurprisingly, encryption did not impede the attack; we successfully logged into the virtualized Windows XP VM using **A3**.

```
...
40115d:   f3 66 a7          repz cmpsw %es:(%edi),%ds:(%esi)
401160:   49                dec    %ecx
401161:   79 13             jns    0x401176
401163:   6a 00             push   $0x0
401165:   68 31 21 40 00    push   $0x402131 ;"Congratulations !!!"
40116a:   68 9a 21 40 00    push   $0x40219a ;
40116f:   6a 00             push   $0x0      ;hWnd
401171:   e8 4a 04 00 00    call   0x4015c0  ;MessageBoxA
401176:   61                popa
401177:   c3                ret
...
```

Figure 8: Binary Code Snippet of Unpacked `crackme.exe`.

Tampering with `crackme.exe`. For this experiment, we acquired a binary program `crackme.exe`[1] from crackmes.de. Figure 8 shows a partial disassembly (after UPX unpacking).

The program asks users to enter a serial number, and displays a "congratulations!!!" message window if the number entered is correct. Our goal is to make the binary code show this message. The traditional attack strategy is to break the serial computation algorithm and write a key generator algorithm. However, our VMI facilitates the much easier alternative of patching the instruction code. In particular, the boxed `jns0x401176` instruction in Fig. 8 at memory address `0x401161` decides whether the entered serial number is correct. To bypass this check, we can either change the machine code `79 13` (`jns 0x401176`) to `78 13` (`js 0x401176`), or `90 90` (NOP them). Both approaches are successful.

Summary. The first case study demonstrates that with no reverse engineering and just some general knowledge of the victim program, our VMI can be employed to successfully compromise Windows authentication protections. With a very small amount of reverse engineering, we are able to also bypass a software piracy check.

6. DISCUSSIONS AND IMPLICATIONS

Our experiments indicate that semantic gap-bridging technologies, as exhibited by our VMI attack, have significantly eased the burden of implementing low-resource attacks against software authentication protections. With only superficial reverse engineering effort and some simple configuration steps, a context-aware, reactive VMI quickly and easily compromises both Linux and Windows authenticators. These results challenge several outdated assumptions and conventional wisdoms about low-resource attacker capabilities.

[1] MD5 checksum: `8f59900d9a5e4fa4ab21115d38e82954`

First, our VMI attack challenges the trust that application programmers usually have in the underlying hardware. In general, application programmers tend to assume that once a program is compiled, its binary code will consistently behave in accordance with the hardware specifications. However, with the introduction of a virtualizaiton-layer [15], such trust is misplaced. Because VMMs virtualize all of a system's hardware resources to the OS (including the instructions for emulation-based virtualization), attackers can now easily *monitor, intercept*, and *tamper* with any of these "hardware" resources.

Second, our results show that safely outsourcing VM images to clouds requires complete trust in cloud providers. Encrypting the VM images (e.g., via FDE [18, 32]) does not prevent a malicious cloud provider from using VMI to bypass the authentication and gain decrypted access to the image's contents. While pre-boot authentication is a possible defense, it is technically challenging to deploy pre-boot authentication in a remote cloud environment, since pre-boot passwords cannot typically be entered remotely. Moreover, even pre-boot authentication can potentially be compromised if it stores cryptographic keys in memory at runtime. Memory forensics techniques such as Fireware [4] or ColdBoot [17] can be used to extract such keys, since the decryption key must exist in main memory in order for both the OS and application data to be decrypted during the VM execution. Therefore, the research community must look for new techniques to protect cloud VMs.

Finally, our results also have implications for the practical use of FDE on mobile devices. Today, many organizations require FDE on laptops to guard against data breaches from lost or stolen computers [36]. FDE products like TrueCrypt [41] work as follows: The user first logs into a mini-OS through a pre-boot authentication, and if the password matches then FDE decrypts and loads the main OS. The user must next enter the password for the main OS to log into the system.

However, in practice, the pre-boot authentication often gets disabled for a number of technical and non-technical reasons. First, end-users must type two passwords to enter the system, which is cumbersome. Second, it often impedes software updates requiring a reboot (though there is a log-in-once option in many FDE products to assist with this [32]). Third, remote users (such as the VM users of IaaS clouds, or remote assistance users) cannot use the system after a reboot until the local user logs in, since they cannot type the pre-boot password remotely. Consequently, many organizations skip pre-boot authentication, and directly use the authentication credentials of the main OS to protect the system. As demonstrated in our attack, such practices are completely insecure. We therefore strongly advocate pre-boot authentication for effective FDE protection.

7. RELATED WORK

Virtualization-based attacks. Our system authentication subversion is related to the Bluepill attack [37] (and the closely related Vitriol [47]), and SubVirt [26] (and the closely related SubXen [44]). These all share the similar philosophy of launching malicious attacks from the virtualization layer. However, they have substantial differences.

First, both Bluepill/Vitriol and SubVirt/SubXen are virtualization-based rootkits that subvert third-party virtualizations. Thus, they must carefully avoid detection by the virtualization. In contrast, our VMI attack explores the relative freedom that a malicious virtualization owner has to easily launch stealthy attacks against the virtualized software.

Second, Bluepill/Vitriol is an ultra thin hypervisor, which means its code base (payload) is typically small. In contrast, our VMI is an ultra fat hypervisor that bridges the semantic gap and provides malicious APIs or configurable interfaces for launching attacks.

Third, SubVirt and SubXen demonstrate the implementation of *additional* malicious backdoor services (e.g., a keystroker sniffer, a phishing web server, and a user sensitive data harvester [26]), but not attacks that enable users to bypass authentication from the hypervisor layer. Unlike our VMI, the attacks are limited to self-contained services that do not attempt to bridge the semantic gap, which has been recognized as the grand challenge for implementing services at the hypervisor layer [5].

Virtual machine introspection. Our attack is also closely related to VMI. Earlier introspection techniques were purely developed for the read-only inspection without affecting the guest-OS states. Some notable examples include Livewire [13], Antfarm [21], XenAccess [33], VMwatcher [19], Ether [8], Virtuoso [9], and VMST [11]. Recently, there were also reactive VMIs proposed. Notable examples include IntroVirt [23] that validates vulnerability-specific predicates at the VMM level to detect and respond to an intrusion, Manitou [29] that makes a corrupted instruction page non-executable when it detects instruction page mismatches, and Exterior [12] that repairs kernel rootkit damages when their footprints are detected.

In addition, there are also some active introspection techniques, such as Lares [34] and SIM [39], but they require modifying the guest OS. Guest OS modification has also been used to realize techniques that mislead the VMI (e.g., [3]).

8. CONCLUSION

We have presented the design, implementation, and evaluation of a context-aware, reactive virtual machine introspection technique that can be leveraged to break the authentication mechanisms of Linux and Windows operating systems with relatively little effort. The attacks use binary analysis to identify critical decision points where authentication succeeds or fails at the binary level, and then leverage a context-aware, reactive VMI to transparently corrupt the control-flow or data-flow of the victim software at that point, resulting in successful authentication without any password-guessing or encryption-cracking. Depending on the instrumentation granularity, our attack can be implemented using hardware virtualization to tamper with syscall related data, or using software virtualization to tamper with register and memory contents, or instruction operands and opcodes, without detection by the guest-OS. Our results indicate that the approach is applicable to a broad class of authentication implementations, practical against real-world OSes and VM images, and useful for both malicious attacks and forensics analysis of virtualized systems and software.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. We are also grateful to Shuo Chen, Kevin Hulin, and Kenneth A. Miller for their feedback on an earlier draft of the paper. This research was supported in part by a research gift from VMware, Inc. and National Science Foundation grant #1054629. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and not necessarily of VMware or the NSF.

References

[1] N. Amit, M. Ben-Yehuda, D. Tsafrir, and A. Schuster. vIOMMU: Efficient IOMMU emulation. In *Proceedings of the USENIX Annual Technical Conference*, 2011.

[2] Apache. Apache HTTP server benchmarking tool. http://httpd.apache.org/docs/2.2/programs/ab.html.

[3] 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. In *Proceedings of the 29th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 82–91, 2010.

[4] A. Boileau. Hit by a bus: Physical access attacks with firewire. Ruxcon, 2006.

[5] P. M. Chen and B. D. Noble. When virtual is better than real. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS)*, pages 133–138, 2001.

[6] E. J. Chikofsky and J. H. C. II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[7] Defense Science Board. Report of the Defense Science Board Task Force on High Performance Microchip Supply. http://www.acq.osd.mil/dsb/reports/ADA435563.pdf, February 2005.

[8] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, pages 51–62, 2008.

[9] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the 32nd IEEE Symposium on Security & Privacy (S&P)*, pages 297–312, 2011.

[10] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *Proceedings of the 24th IEEE Symposium on Security & Privacy (S&P)*, 2003.

[11] Y. Fu and Z. Lin. Space traveling across VM: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *Proceedings of the 33rd IEEE Symposium on Security & Privacy (S&P)*, 2012.

[12] Y. Fu and Z. Lin. EXTERIOR: Using a dual-VM based external shell for guest-OS introspection, configuration, and recovery. In *Proceedings of the 9th Annual International Conference on Virtual Execution Environments (VEE)*, pages 97–110, 2013.

[13] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the 10th Annual Network & Distributed System Security Symposium (NDSS)*, 2003.

[14] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 193–206, 2003.

[15] R. P. Goldberg. *Architectural Principles of Virtual Machines*. PhD thesis, Harvard University, 1972.

[16] K. Griffin, S. Schneider, X. Hu, and T.-C. Chiueh. Automatic generation of string signatures for malware detection. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 101–120, 2009.

[17] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold-boot attacks on encryption keys. In *Proceedings of the 17th USENIX Security Symposium*, 2008.

[18] Help Net Security. Encrypt and protect virtual machine images. http://www.net-security.org/secworld.php?id=11825, 2011.

[19] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, pages 128–138, 2007.

[20] N. M. Johnson, J. Caballero, K. Z. Chen, S. McCamant, P. Poosankam, D. Reynaud, and D. Song. Differential slicing: Identifying causal execution differences for security applications. In *Proceedings of the 32nd IEEE Symposium on Security & Privacy (S&P)*, pages 347–362, 2011.

[21] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Antfarm: Tracking processes in a virtual machine environment. In *Proceedings of the USENIX Annual Technical Conference*, 2006.

[22] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. VMM-based hidden process detection and identification using lycosid. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, pages 91–100, 2008.

[23] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 91–104, 2005.

[24] J. Kephart and W. Arnold. Automatic extraction of computer virus signatures. In *Proceedings of the 4th Virus Bulletin International Conference*, pages 178–184, 1994.

[25] G. H. Kim and E. H. Spafford. The design and implementation of Tripwire: A file system integrity checker. In *Proceedings of the 1st ACM Conference on Computer and Communications Security (CCS)*, pages 18–29, 1994.

[26] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. SubVirt: Implementing malware with virtual machines. In *Proceedings of the 27th IEEE Symposium on Security & Privacy (S&P)*, pages 314–327, 2006.

[27] S. T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou. Designing and implementing malicious hardware. In *Proceedings of the 1st USENIX Workshop on Large-scale Exploits and Emergent Threats (LEET)*, 2008.

[28] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 17th Annual Network & Distributed System Security Symposium (NDSS)*, 2010.

[29] L. Litty and D. Lie. Manitou: A layer-below approach to fighting malware. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability (ASID)*, pages 6–11, 2006.

[30] Memcached. Memcached: a distributed memory object caching system. http://memcached.org.

[31] Microsoft TechNet. BitLocker drive encryption technical overview, 2008. http://technet.microsoft.com/en-us/library/cc732774.aspx.

[32] R. Mogull. FireStarter: Is full disk encryption without pre-boot secure? https://securosis.com/blog/firestarter-is-full-disk-encryption-without-pre-boot-secure, 2010.

[33] B. D. Payne, M. Carbone, and W. Lee. Secure and flexible monitoring of virtual machines. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC)*, 2007.

[34] B. D. Payne, M. Carbone, M. I. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *Proceedings of the 29th IEEE Symposium on Security & Privacy (S&P)*, pages 233–247, 2008.

[35] J. Pfoh, C. Schneider, and C. Eckert. Nitro: Hardware-based system call tracing for virtual machines. In *Proceedings of the 6th International Conference on Advances in Information and Computer Security (IWSEC)*, pages 96–112, 2011.

[36] L. Ponemon. Airport insecurity: The case of missing and lost laptops. Technical report, Ponemon Institute, 2008.

[37] J. Rutkowska. Introducing Blue Pill, June 2006. http://theinvisiblethings.blogspot.com/2006/06/introducing-blue-pill.html.

[38] Seagate. Maxtor basics personal storage 3200 (PS 3200) virus. http://knowledge.seagate.com/articles/en_US/FAQ/205131en.

[39] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi. Secure in-VM monitoring using hardware virtualization. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, pages 477–487, 2009.

[40] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proc. of USENIX Annual Technical Conference*, 2001.

[41] TrueCrypt. TrueCrypt: Free open-source on-the-fly encryption. http://www.truecrypt.org, 2012.

[42] VMware. VMware vCenter converter. http://www.vmware.com/products/converter, 2013.

[43] S. Vogl and C. Eckert. Using hardware performance events for instruction-level monitoring on the x86 architecture. In *Proceedings of the 5th European Workshop on System Security (EuroSec)*, 2012.

[44] R. Wojtczuk. Subverting the Xen hypervisor. In *Black Hat Technical Security Conference*, 2008.

[45] V. Yegneswaran, J. T. Giffin, P. Barford, and S. Jha. An architecture for generating semantics-aware signatures. In *Proceedings of the 14th USENIX Security Symposium*, 2005.

[46] F. Zhang, J. Chen, H. Chen, and B. Zang. CloudVisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 203–216, 2011.

[47] D. A. D. Zov. Hardware virtualization rootkits, July 2006. http://www.theta44.org/software/HVM_Rootkits_ddz_bh-usa-06.pdf.