Guest-Transparent Instruction Authentication for Self-Patching Kernels

Dannie M. Stanley, Zhui Deng, and Dongyan Xu
Department of Computer Science
Purdue University
West Lafayette, IN 47907
Email: {ds,deng14,dxu}@cs.purdue.edu

Rick Porter
Applied Communication Sciences
Piscataway, NJ 08854
Email: fporter@appcomsci.com

Shane Snyder
US Army CERDEC
Information Assurance Division
Aberdeen Proving Ground, MD

***Abstract*—Attackers can exploit vulnerable programs that are running with elevated permissions to insert kernel rootkits into a system. Security mechanisms have been created to prevent kernel rootkit implantation by relocating the vulnerable physical system to a guest virtual machine and enforcing a $W \oplus KX$ memory access control policy from the host virtual machine monitor. Such systems must also be able to identify and authorize the introduction of known-good kernel code. Previous works use cryptographic hashes to verify the integrity of kernel code at load-time. The hash creation and verification procedure depends on immutable kernel code. However, some modern kernels contain self-patching kernel code; they may overwrite executable instructions in memory *after* load-time. Such dynamic patching may occur for a variety of reason including: CPU optimizations, multiprocessor compatibility adjustments, and advanced debugging. The previous hash verification procedure cannot handle such modifications. We describe the design and implementation of a procedure that verifies the integrity of each modified instruction as it is introduced into the guest kernel. Our experiments with a self-patching Linux guest kernel show that our system can correctly detect and verify all valid instruction modifications and reject all invalid ones. In most cases our patch-level verification procedure incurs only nominal performance impact.**

I. Introduction

Kernel rootkits are a particularly virulent kind of malicious software that infects the operating system (OS) kernel. They are able to create and execute code at the highest OS privilege level giving them full control of the system. They are not only able to carry out malicious actions but are also able to evade detection by modifying other system software to hide their evidence. Once detected a kernel rootkit can be difficult to remove; it is not always obvious the extent to which the system has been modified. The goal therefore is to *prevent* kernel rootkit infection.

Attackers can exploit vulnerable programs that are running with elevated permissions to insert kernel rootkits into a system. Security mechanisms like NICKLE have been created to prevent kernel rootkits by relocating the vulnerable physical system to a guest virtual machine and enforcing a $W \oplus KX$ memory access control policy from the host virtual machine monitor (VMM)[1]. The $W \oplus KX$ memory access control policy guarantees that no region of guest memory is both writable *and* kernel-executable.

The guest system must have a way to bypass the $W \oplus KX$ restriction to load valid kernel code, such as kernel drivers, into memory. To distinguish between valid kernel code and malicious kernel rootkit code, NICKLE[1] and others [2][3] use cryptographic hashes for verification. Offline, a cryptographic hash is calculated for each piece of valid code that may get loaded into the guest kernel. Online, the VMM intercepts each guest attempt to load new kernel code and calculates a hash for the code. If the online hash matches an offline hash, the load is allowed.

Some modern kernels however, are "self-patching;" they may patch kernel code at run-time. If the patch is applied *prior* to hash verification, then the hashes will not match. If the patch is applied *after* hash verification, then the memory will be read-only, due to $W \oplus KX$ enforcement, and the patch will fail. Such run-time patching may occur for a variety of reason including: CPU optimizations, multiprocessor compatibility adjustments, and advanced debugging. The previous hash verification procedure cannot handle such modifications.

We describe the design and implementation of a system that verifies the integrity of each instruction introduced by a self-patching kernel. We verify each instruction by comparing it to a whitelist of valid instruction patches. We generate the whitelist ahead of time while the guest is offline. When online, certain predetermined guest events such as write-faults and code loading, will trigger a trap into the host and give our system the opportunity to verify new instructions.

Our system is guest-transparent; no modifications to the guest operating system are required. However, the whitelist construction is dependent on the guest kernel. Each guest kernel may patch itself in different ways. As we describe in Section IV, the whitelist creation procedure requires knowledge of the guest kernel to collect all of the possible valid patches. We discovered that the Linux kernel has six different facilities that influence code modification. We describe each

[0]The research reported in this paper was performed in connection with Contract Number W15P7T-11-C-A021 with the US Army CECOM LCMC Command. The views and conclusions contained in this paper are those of the authors and should not be interpreted as presenting the official policies or position, either expressed or implied, of the US Army CECOM LCMC Command or the US Government unless so designated by other authorized documents. Citation of manufacturers or trade names does not constitute an official endorsement or approval of the use thereof. The US Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

facility and its impact on whitelist creation in Section V.

II. Related Work

The implementation of our instruction authentication procedure, described in section V, is part of our reimplementation of NICKLE[1]. NICKLE is a guest-transparent virtual machine monitor (VMM) that prevents unauthorized code from executing with kernel-level privileges. NICKLE, as originally implemented, does not take advantage of the hardware-assisted virtualization extension present in many current processors. Our implementation, named NICKLE-KVM, takes advantage of these extensions by adding NICKLE functionality to the Linux Kernel-based Virtual Machine (KVM) virtualization infrastructure.

Additionally, the original NICKLE does not support self-patching guest kernels. NICKLE-KVM, with our instruction authentication subsystem, removes this restriction. Two other systems similar in nature to NICKLE, SecVisor[2] and hvmHarvard[3], also require that the guest does not contain self-patching kernel code.

III. Outline

The remaining content of this paper is structured as follows: In Section IV, we outline the problem and describe the design of our solution. Section IV-A describes the context of the problem and provides a motivating example. In Section IV-B we describe our solution in detail and demonstrate how it provides a remedy to our motivating example. Section V describes the implementation of our system and we report our experimental results in Section VI. Finally, in Section VII, we recap our findings and suggest some future work.

IV. System Design

A. Problem

Preventing kernel rootkit injection is a challenge. Software is often buggy. If a bug is exploited in kernel mode, then malicious software can gain a foothold into the system with maximum privilege. Once such privilege is attained, the attacker can effectively disable all other protections. For example, like NICKLE, some operating systems implement a $W \oplus X$ memory access control scheme for kernel code. Unfortunately, a $W \oplus X$ policy cannot be reliably enforced by the same system that it is trying to protect. Because the operating system has full control over the hardware, a bug may allow an attacker to disable all protections. Systems like NICKLE solve this problem by relocating the vulnerable physical system to a *guest* virtual machine (VM) and providing $W \oplus KX$ access control policy enforcement from the *host* virtual machine monitor (VMM). This relocation enables such systems to provide services *below* the operating system [4]. In particular, the host can strictly control guest access to memory.

As part of enforcing a $W \oplus KX$ policy in the guest, the host must set all regions of memory containing guest kernel code to read-only and executable. To determine the addresses of these regions, the host can trap on guest events that introduce new kernel code into the guest. Two such events occur at guest *kernel load-time* and *kernel module load-time*. To trap on these events, the guest functions that load code into the kernel can be intercepted through, for example, debug breakpoints. Breakpoint execution triggers a VM exit enabling the host to extract the address range of the guest's newly loaded code. Once the host determines the regions of memory containing new kernel code, the code can be verified using cryptographic hashes. Following verification, the code memory regions can be then set to read-only and executable.

1) Code Authentication: Code loaded into kernel space originates in executable files stored on disk. Executable file formats divide executable files into sections. At least one section contains machine instructions. The most primitive executable file contains one primary section that holds the machine instructions. We will refer to this section as the *text section*. In addition to the text section, some executable files have many other sections that contain machine instructions. We discovered 11 such unique sections during our experimentation (see Section V for details). We refer to all sections of a single executable file that contain machine instructions as *executable texts*.

To authenticate kernel code as it is introduced into the kernel space, the original NICKLE uses cryptographic hashes. When the guest is offline, a cryptographic hash is calculated for each executable file that may get loaded into the guest. The hash is calculated over the executable text of the file. Each hash is stored, along with other meta data, in a database. This hash database is stored in the host and is inaccessible from the guest. When the guest is online and an attempt to load new kernel code into the guest is detected, a trap to the host occurs. The host then calculates a hash over the executable texts as they appear in the guest's memory space. If the hash matches the hash of the corresponding executable file, then the code addition is allowed and the permissions for the new executable texts are set to read-only. If the hash does not match, then the code addition is not allowed.

Comparing offline and online hashes does not work for relocatable code, such as the kernel and kernel modules, since the relocation process changes addresses in the online code. NICKLE works around this problem by writing zeroes to all relocation call-sites prior to calculating hashes.

2) Code Modification: NICKLE's hash creation and verification procedure depends on immutable kernel code. However, some modern kernels are "self-patching;" they may overwrite executable instructions in memory *after* load-time. The original NICKLE hash verification procedure cannot handle such modifications.

As a motivating example, Linux kernel versions greater than 2.5.68 include support for "alternative instructions" (altinstructions[1]) [5]. Altinstructions enable the kernel to optimize code at run-time based on the capabilities of the CPU. The example provided in the original kernel mailing list announcement was for "run-time memory barrier patching" (RTMBP). According

[1]We adopt this label to alleviate confusion between the intuitive meaning of the phrase "alternative instructions" and the Linux facility.

to the RTMBP announcement the default Linux memory barrier instruction sequence, `lock; addl $0,0(%%esp)`, is slow on Pentium 4 processors and should be replaced with the processor's built-in memory barrier instruction: `lfence`. To take advantage of the CPU's capabilities, the Linux distributor could package a separate kernel image tailor-made for the Pentium 4 processor or dynamically patch the instructions at run-time using the altinstructions kernel feature.

Consider how the existing code authentication procedure would work in the presence of RTMBP. Offline a hash would be calculated over a section of text containing the unoptimized memory barrier: `lock; addl $0,0(%%esp)`. When online and that section of code gets loaded into the guest kernel, the host has two options. The first option is for the host to check the hash prior to run-time modifications. If the corresponding memory is set to read-only immediately following hash verification, then future run-time modifications will fail and the guest system will run without optimizations (because the destination kernel code region is set to read-only). The second option is for the host to check the hash after run-time modifications have been applied. However, the hash will not match on a Pentium 4 processor because the stored hash was calculated over the original instruction (`lock; addl $0,0(%%esp)`) not the optimized instructions (`lfence`).

B. Approach

Our approach to solving the problem introduced by self-patching kernels is *patch-level verification*. We observe that if the guest kernel can introduce new instructions at run-time and those instructions take the form of a constrained set of patches (possibly requiring source code or vendor documentation to determine), then we can create a whitelist of valid instruction patches.

We refer to each valid replacement instruction as a "patch" and each description of the patch as a "patch definition." Offline we generate a whitelist of valid patch definitions which we refer to as the "patch set." Each patch definition is a 3-tuple: (patch-location, patch-length, patch-data). The *patch-location* describes the address, relative to the start of the text section, where the patch may get applied. The *patch-length* describes the size of the replacement instruction. The *patch-data* holds the replacement instruction in raw binary form. We refer to the location in a text section which may or may not be patched at run time as a "patch site."

The patch set and the hash database are alike; the hash database is used to verify the integrity of new executable kernel code (minus patches) and the patch set is used to verify individual patches to kernel code. The patch set is created while the guest system is offline, stored in the host system, and is inaccessible from the guest.

1) Patch Set Creation: Patch set creation varies widely depending on the self-patching mechanisms present in the guest kernel. During our experimentation with a Linux guest kernel, we discovered six unique facilities with which the kernel patches itself. We list them in Section V and describe some of the challenges they pose to patch-level verification. For now, we reintroduce our motivating example of RTMBP to demonstrate the creation of our patch set.

Recall that RTMBP uses the altinstructions mechanism to achieve kernel patching. Prior to getting loaded into the system, kernel code is stored on disk in executable files; in this case, kernel code is stored in an executable and linkable format (ELF) file. Each ELF file carries with it all of its own altinstruction patches defined by two ELF file sections: ".altinstructions" and ".altinstr_replacement". The .altinstructions section contains a set of zero or more "alt_instr" structures with the following fields: (*instr, *replacement, cpuid, instrlen, replacementlen). Two fields of this structure, **instr* and *instrlen*, map directly to our patch definition fields patch-location and patch-length respectively. The third patch definition field, patch-data, can be copied directly from the bytes stored at **replacement* (of length *replacementlen*). The *replacement pointer points to an address located in the .altinstr_replacement ELF section.

Our patch set creation procedure adds two patch definitions for each alt_instr. The first patch definition corresponds to the default instruction, such as `lock; addl $0,0(%%esp)` for RTMBP, located in the text. The second patch definition corresponds to the replacement instruction, `lfence` for RTMBP, described by the alt_instr. Both definitions have the same patch-location values and the same patch-length values because they are both candidates for the same patch site. When patch-level verification occurs at run-time, the instruction present in memory must match one of the candidates.

Each executable file (kernel or kernel module) that may get loaded into the guest kernel has a corresponding offline patch set. When the guest comes online and attempts to load code from an executable file, the corresponding offline patch set entries are copied by the host into the online patch set. The online patch set is identical in structure to the offline patch set. The patch-location values however are recalculated to reflect the guest memory address of the patch site (as opposed to the relative patch-location in the original executable file). The online patch set is a subset of the offline version. It only contains entries for texts that are loaded into the guest kernel.

2) Run-Time Patch-Level Verification: If the guest is protected by a $W \oplus KX$ security policy then during run-time when the guest self-patching kernel tries to modify previously authenticate code, an access violation will occur (write-fault). This access violation can be trapped by the host and used as an opportunity to perform patch-level verification.

The write-fault event provides an address corresponding to the access control violation. For each write-fault we lookup the faulting address in our online patch set. If the instructions to-be-written match the patch-data field of one of the patch definitions in the patch set, then verification succeeds.

If patch-level verification succeeds, then the write is permitted. We temporarily remove the read-only restriction allowing the guest to execute one write instruction, the one that triggered the write fault, and then we reapply the read-only access restriction. This procedure is illustrated in Figure 1.

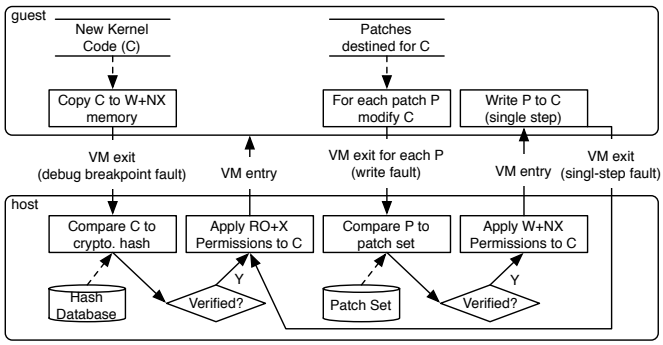

Fig. 1. Run-Time Patch-Level Verification

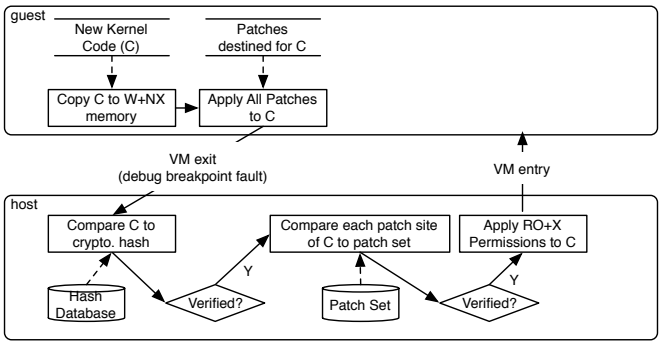

Fig. 2. Load-Time Optimized Patch-Level Verification

3) Load-Time Optimized Patch-Level Verification: For our system, each write-fault causes two VM exits. The first is due to the write-fault itself, which is unavoidable for systems like NICKLE in the presence of a self-patching kernel. The second VM exit allows us to reapply the read-only access control restriction to kernel code. VM exits are expensive, therefore we offer the following optimization which significantly reduces the number of VM exits caused by self-patching kernels.

During experimentation we discovered that many changes made by self-patching kernels happen immediately after the code is loaded into memory and before it is executed. Recall from Section IV-A2 that if we try to compare the cryptographic hash of the code *after* it has been modified, then the hashes will not match. However, we can adjust the hash generation procedure to accommodate self modifying code. Recall from Section IV-A1 how NICKLE calculates a cryptographic hash in the presence of relocatable code; it writes all zeroes to the relocation sites prior to hash calculation. We use a similar method for load-time optimized patch-verification. From the patch set we know all of the patch sites present in the code. We write zeroes to those locations prior to hash creation. When online, we *defer* verification until after the code has been loaded and modified.

At verification time we first check the integrity of the code using the hash database. Secondly we check the integrity of each unique patch site using the corresponding patch set. If the hash matches and the contents of each patch site are verified, then patch-level verification succeeds and the code is added to authenticated memory. Figure 2 demonstrates this procedure in the context of NICKLE-KVM. As shown in the figure, the optimized approach incurs no additional VM exits for patch-level verification.

V. System Implementation

Our patch-level verification procedure is implemented as a subsystem of NICKLE-KVM. NICKLE-KVM is a NICKLE-like system based on KVM. KVM is a Linux-based VMM that takes advantage of hardware-assisted virtualization. The original NICKLE implementation, based on QEMU, does not take advantage of hardware-assisted virtualization. Instead, it uses binary translation to intercept each instruction to preserve the NICKLE guarantee: "No unauthorized code can be executed at the kernel level.[1]" Instead of the instruction-level redirection technique used by NICKLE, we use the page-level redirection technique introduced by hvmHarvard which takes advantage of hardware-assisted virtualization for better performance [3]. In the absence of a complete description of NICKLE-KVM, we describe the parts we leveraged for our implementation next.

To evaluate our design, we implemented the optimized patch-level verification solution described in Section IV-B3 for both kernel and module loading. To support our solution, we modified NICKLE-KVM in the following ways:

1) At both kernel load-time (boot) and module load-time the guest must pass control to the host for code authentication. Therefore, at guest system start up we set a hardware debug break point on the return addresses of the guest Linux functions named "init_post" and "trim_init_extable" for kernel and module authentication respectively. These addresses are reached immediately after patches have been applied making them an ideal point in the loading sequence to trigger our authentication procedures. KVM gives us access to the guest's virtual CPU (vcpu) allowing us to set the corresponding debug registers for these break points. Our implementation catches the resulting debug exceptions and then performs code and patch-level verification.
2) If code verification succeeds, NICKLE-KVM must apply the read-only and execute permissions to the new code prior to returning control to the guest. We notify NICKLE-KVM of the new code and it handles the manipulation of and enforcement of the page-level permissions.
3) If code verification fails, the host can take protective measures. In the case of kernel verification failing, the host could force the guest to shutdown. In the case of module verification failing, the host could force the module to fail loading by manipulating guest execution. For example, in one version of our implementation we forced the return value of the function "module_finalize" to be -1 by manipulating a guest vcpu register (EAX).

A. Experimental Setup

We implemented NICKLE-KVM and our patch-level verification subsystem using an Intel i5-2410M 2.30GHz processor. Both our host and guest systems run an *unmodified* version of Ubuntu (11.04 2.6.38-8-generic). The host runs the 64 bit version of Ubuntu (x86_64) and the guest runs the 32 bit version (i686). NICKLE-KVM is a modified version of KVM version 2.6.38.6.

B. Patch Verification

Recall that patch set creation is heavily dependent on knowledge of the guest kernel. Through analysis of the Linux kernel source code we found six facilities that contribute to kernel run-time patching: Alternative Instructions, SMP Locks, Jump Labels, Mcounts, Paravirtual Instructions, and Kprobes. For our implementation four of the six kinds apply, however we list them all here for completeness. For each of the four that do apply, we describe how it influences patch set creation.

1) Alternative Instructions: Our motivating example of RTMBP, introduced in Section IV, was an instance of altinstructions. Recall that the purpose of altinstructions is to allow the kernel to optimize code at run-time based on the capabilities of the CPU. The ELF file that contains the loadable kernel code has two sections used by kernel patching functions related to altinstructions: .altinstructions and .altinstr_replacement. The details of how we derive a patch definition from these sections can be found in Section IV-B1. The altinstructions patch sites do not change after the code has been loaded for the first time because the the CPU capabilities do not change during run-time.

In all cases that we encountered patching facilities modified only the .text ELF section with the one exception of altinstructions getting applied to the kernel (not modules). In this case, altinstructions were applied to both the .text *and* .init.text executable sections.

2) SMP Locks: The SMP locks code modification mechanism is similar to altinstructions. Based on the capabilities of the CPU, the kernel module code may be modified at load-time. The Linux kernel modules shipped with Ubuntu are compiled for both symmetric multiprocessing (SMP) and uniprocessor systems (UP). When the code is running on an SMP system, a lock prefix (0xF0 for x86) is used to indicate that the following read-modify-write instructions should be executed atomically. However, when the kernel is running on a UP system the SMP locks are not needed. In that case, the kernel modifies the module code to remove SMP locks (by replacing each with DS segment override prefix, 0x3E for x86[2])

ELF files with SMP locks in their code have a special section named ".smp_locks" which holds zero or more 4-byte addresses. Each address corresponds to an SMP lock site in the text section. The length of the lock is one byte. Therefore for each SMP patch site, two patch definitions are created: (<address from header>, 1, 0xF0) and (<address from header>, 1, 0x3E). For the same reasons as altinstructions, the module's SMP locks code does not change after the module has been loaded.

3) Jump Labels: Jump labels were introduced into the Linux kernel to optimize kernel tracing[6]. Prior to Jump Labels, if a developer wanted to add a trace point to his code he included a conditional statement to evaluate if tracing was enabled. If enabled, the code would jump to a code block that provided tracing information. To avoid the overhead associated with the conditional statement, Jump Labels are used. Essentially a Jump Label site in code contains either a jump instruction (JMP) or a NOP instruction. At module load-time, the site contains a JMP instruction and it is overwritten with a NOP instruction. When tracing is enabled for that Jump Label, the JMP instruction is reintroduced. The result is that most of the time tracing logic incurs no overhead (except that which is associated with the NOP instruction).

ELF files with Jump Labels in their code have a special section named "__jump_table." For each Jump Label an entry exists in the __jump_table table. Each entry has the following fields: *code*, *target*, *key*. The code field corresponds to the location in text where the Jump Label is inserted. The target field corresponds to the jump target (where to jump when tracing is enabled for this Jump Label). Each Jump Label is identified by a unique key. When one wants to enable tracing for that Jump Label he calls the Linux function named "enable_jump_label" with the Jump Label key as a parameter.

For each Jump Label location, two patch definitions are created: (<address from code field>, 5, <NOP of size 5[3]>) and (<address from code field>, 5, JMP <address from label field>). Unlike altinstructions and SMP Locks, Jump Label code sites get changed at load-time (NOP inserted) and when tracing is enabled (JMP inserted). For this implementation we are interested in the changes made at load-time (NOP inserted). For kernel tracing to work, NICKLE-KVM must perform patch-level verification at write-fault time as described in Section IV-B2.

4) Mcounts: When Linux kernel code is compiled with the "-pg" flag the compiler automatically adds a call to the mcount procedure at the beginning of each kernel function [7]. This feature enables kernel profiling and tracing. To optimize Mcounts, when code is loaded that contains mcount call sites, the kernel automatically replaces the call instructions with a DS segment override prefixes (0x3E) like in the case of SMP Locks. Later when mcount tracing is enabled, the call instruction will be restored.

Because the mcount symbol is relocatable, the patch-location can be calculated based on relocation information found in the .rel.text section. The start of the patch-location is one byte before the mcount symbol address in the text. For each Mcount patch site, two patch definitions are created: (<byte offset of mcount from .rel.text - 1>, 1, 0x3E) and (<byte offset of mcount from .rel.text - 1>, 1, 0xe8). Like Jump Labels, mcount call sites will get changed during run-

[2]See the Linux kernel function alternatives_smp_unlock() for details.

[3]0x3e 0x8d 0x74 0x26 0x00

time when tracing is enabled. For kernel tracing to work, NICKLE-KVM must perform patch-level verification at write-fault time as described in Section IV-B2.

5) Other Facilities: During our exploration of the Linux kernel source code we discovered two facilities that influenced run-time patching but did not ultimately apply to this implementation: Paravirtual Instructions and Kprobes. We list them briefly here for posterity.

Like the four facilities detailed previously, Paravirtual Instructions have a special section in the ELF file (.parainstructions) [8]. The .parainstructions section has an entry for each location in the text that holds an instruction which needs to be modified if the guest is running in a paravirtualized environment. Our guest is not running in a paravirtualized environment. Therefore, we explicitly turn off this feature in the guest kernel by supplying the "noreplace-paravirt" boot option at boot-time. If paravirtualized instruction were to be used in our implementation, further work would have to be done to generate the corresponding patch set entries.

Kprobes[9] (including jprobes and kretprobes) are distinct from the other facilities in that they do not have a corresponding ELF file section for calculating patch set entries. Kprobes are typically used for debugging and each patch site is defined by the end user at debug time. Therefore, Kprobes could work in the presence of patch-level instruction verification if the end user supplied a list of potential probe patch sites to the offline patch set creation procedure. Because Kprobes are inserted at run-time, they do not apply directly to the optimized implementation that we describe here.

VI. Evaluation

To evaluate our system we generated patch sets for the Linux kernel (vmlinux) and 3308 kernel modules (only 11 modules[4] were needed by our guest system). The kernel contained 31643 patch sites in total. The 11 modules contained 639 patch sites in total. After implementing patch-level verification, NICKLE-KVM correctly verified the integrity of all 32282 patch sites.

If an attacker were to modify the text section of previously profiled code, then verification should fail or the malicious code should be discarded. We verified this scenario by manipulating the text section of a module and loading it into the guest. If the modified instruction was not in a location subject to receiving all zeros, like a patch or relocation site, then our system prevented the module from loading as expected. If the spurious instruction was part of a patch or relocation site, then the module passed the initial hash verification but the kernel overwrote the instruction, the instruction was discarded, and the module was allowed to load. If the spurious instruction was part of a patch site and was overwritten by the kernel, then as in the relocation case, the spurious instruction was discarded and the module loaded without the instruction. If the spurious instruction was part of a patch site and was *not* overwritten by the kernel, then patch-level verification correctly identified the foreign code and prevented the module from loading.

If an attacker were to modify one of the candidate replacement instructions found in the .altinstr_replacement section of previously profiled code, then verification should fail or the malicious code should be discarded. We verified this scenario by manipulating the .altinstr_replacement section of a module and loading it into the guest. If the spurious instruction was part of an instruction that was selected at load time by the guest kernel, then patch-level verification failed and the module was prevented from loading.

Our implementation incurs no additional VM exits for patch-level verification. We merely reuse the single VM exit already required by NICKLE-KVM to perform hash verification. In the presence of patches that are applied well after loading, such as Kprobe patches, the less efficient procedure described in Section IV-B2 is required.

VII. Summary

NICKLE-like systems must have a way to authenticate kernel code when it is loaded into the guest kernel space. Previous NICKLE-like systems were not able to authenticate code introduced by self-patching kernels. Our procedure provides a way for NICKLE-like systems to accommodate self-patching kernels by verifying each patch introduced by the kernel. We implemented our system in the context of NICKLE-KVM and demonstrated how patch-level verification correctly permits valid kernel patches to be applied and rejects patches that are invalid.

[4] 8139cp, 8139too, binfmt_misc, floppy, i2c-piix4, lp, parport, parport_pc, ppdev, psmouse, serio_raw

References

[1] R. Riley, X. Jiang, and D. Xu, "Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing," in *Recent Advances in Intrusion Detection*. Springer, 2008, pp. 1–20.

[2] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 335–350, October 2007.

[3] M. Grace, Z. Wang, D. Srinivasan, J. Li, X. Jiang, Z. Liang, and S. Liakh, "Transparent protection of commodity os kernels using hardware virtualization," *Security and Privacy in Communication Networks*, pp. 162–180, 2010.

[4] P. Chen and B. Noble, "When virtual is better than real [operating system relocation to virtual machines]," in *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, may 2001, pp. 133 – 138.

[5] A. Kleen, "Runtime memory barrier patching," Linux-Kernel Mailing List, April 2003. [Online]. Available: http://lwn.net/Articles/29599

[6] J. Corbet, "Jump label," LWN Article, October 2010. [Online]. Available: http://lwn.net/Articles/412072/

[7] S. Rostedt, "mcount tracing utility," Linux-Kernel Mailing List, January 2008. [Online]. Available: http://lwn.net/Articles/264029

[8] R. Russell, "x86 paravirt_ops: Binary patching infrastructure," Linux-Kernel Mailing List, August 2006. [Online]. Available: http://lwn.net/Articles/194340/

[9] S. Goswami, "An introduction to kprobes," LWN Article, April 2005. [Online]. Available: http://lwn.net/Articles/132196/